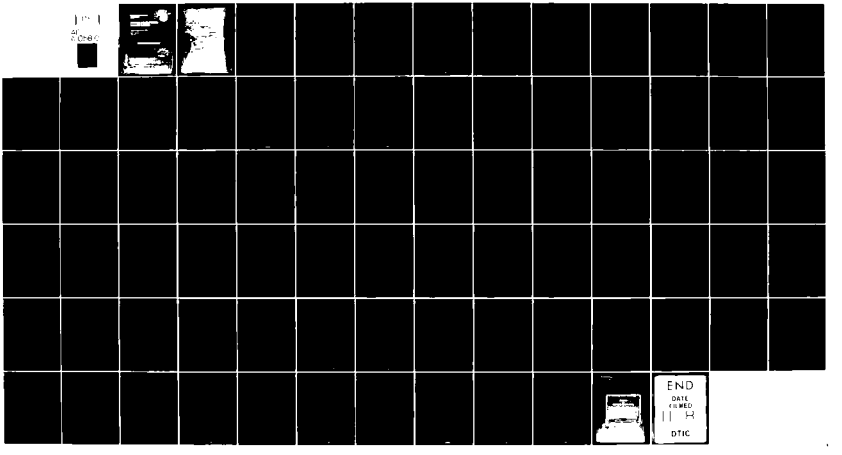
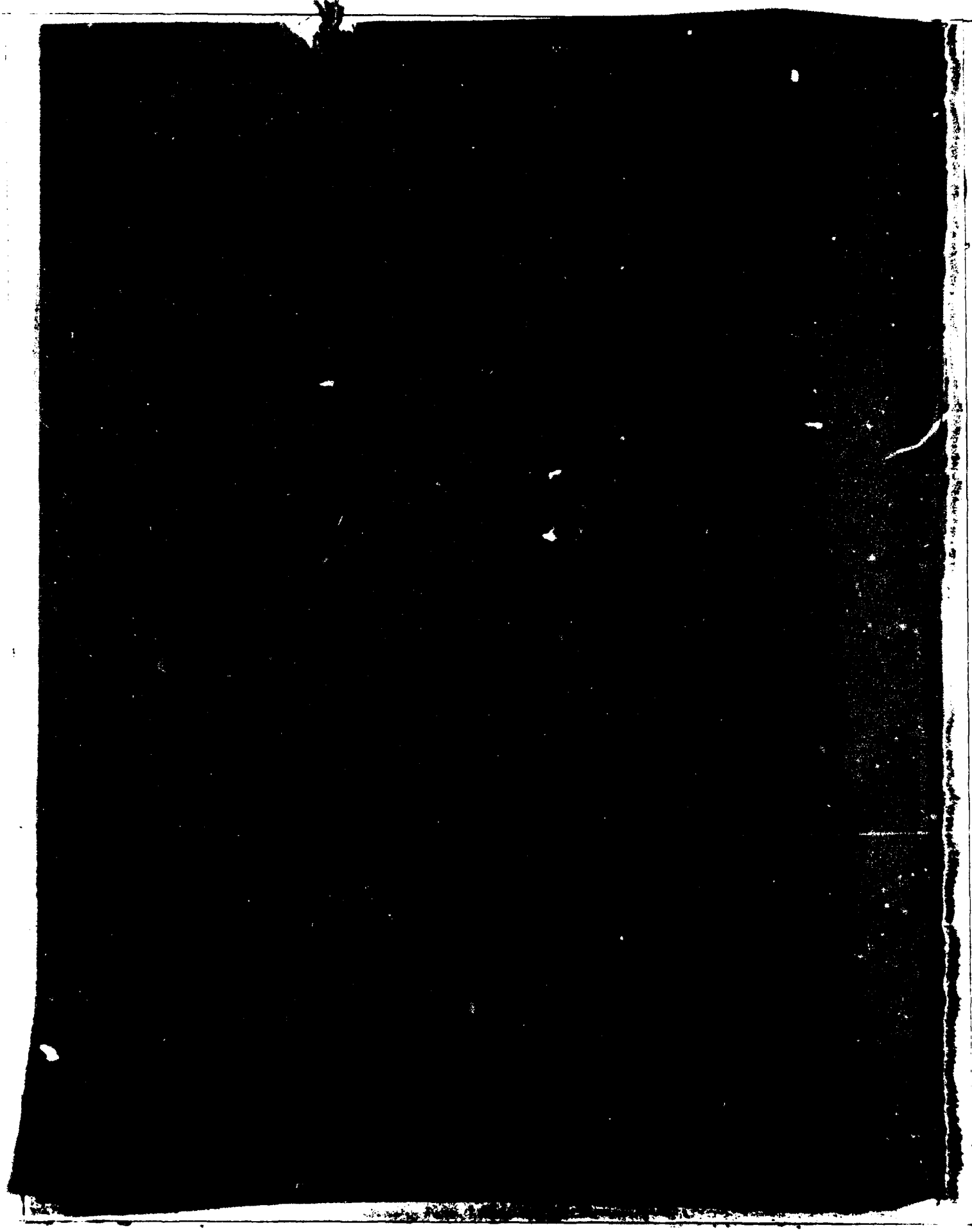
AD-A105 810 POLYTECHNIC INST OF NEW YORK BROOKLYN DEPT OF ELECTR--ETC F/G 9/2
SOFTWARE MODELING STUDIES EXECUTIVE SUMMARY,(U)
JUL 81 M L SHOOMAN, H RUSTON F30602-78-C-0057
UNCLASSIFIED POLY-EE80-006 RADC-TR-81-183-VOL-1 NL





UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-81-183 Vol 1 (of four)	2. GOVT ACCESSION NO.: AD-A105 810
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): SOFTWARE MODELING STUDIES EXECUTIVE SUMMARY	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report Jan 78 — Oct 80
6. PERFORMING ORG. REPORT NUMBER: PCIY-EE 80-006	
7. AUTHOR(s): Martin L. Shooman, Henry Ruston	8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0057
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Polytechnic Institute of New York, 333 Jay Street, Brooklyn NY 11201	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F, 2304J401
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIE), Griffiss AFB NY 13441	12. REPORT DATE: July 1981
	13. NUMBER OF PAGES: 80
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Rocco F. Iuorno (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Engineering
Software Management Models
Software Reliability Models
Test Models
Complexity Measures
Software Errors

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report documents the research performed under RADC Contract by Polytechnic Institute of New York in the area of complexity measures, test models and techniques, methods for developing programs with low error content, software reliability models and software management models. In this volume research described in previous progress reports, technical reports and in Volumes II, III and IV is summarized. Unfinished work, not previously reported, is described. Significant results are highlighted

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A

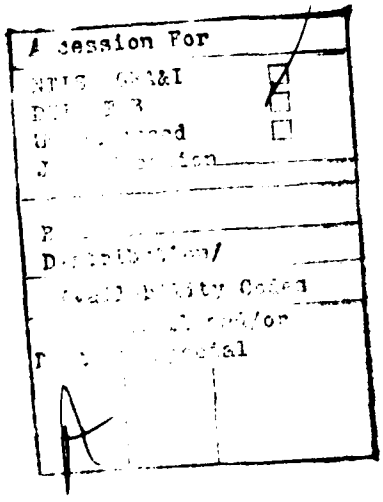
408717

Cont'd

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

with their interrelation and potential.



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

FINAL REPORT

SOFTWARE MODELING STUDIES

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

LIST OF FIGURES

LIST OF TABLES

1.0 Summary and Results

1.1 Objective

This report documents the research performed under Contract No. F30602-78-C-0057 by the Polytechnic Institute of New York for Rome Air Development Center during the period January 1, 1978 to December 31, 1979. Research described in previous progress and technical reports, and in the appended three technical reports is summarized. Unfinished research not previously reported is described. The significant results are highlighted with their interrelations and potential.

1.2 Organization

This final report is composed of four volumes; the present report is labeled Volume 1 and contains a summary of the work performed in the two-year span of the contract. Volumes 2, 3, and 4 are technical reports and are entitled, "The Polynomial Measure of Complexity," "Experimental Study of Two-Dimensional Language vs Fortran for First-Course Programmers," "A Statistical Theory of Computer Program Testing," respectively.

The technical contents are organized into five major subdivisions, Chapters 4.0, 5.0, 6.0, 7.0, and 8.0. Chapter 1.0 and the following two chapters provide a summary, an introduction and discussion of the research performed.

Chapter 4.0 contains a technical summary of the research on complexity measures. Similarly, test models and techniques are described in Chapter 5.0, and program methods for low error content in Chapter 6.0. Chapter 7 gives a view to the work on software reliability models. Chapter 8 describes software management models. Chapter 9 contains brief descriptions of newly initiated and continuing efforts.

Chapter 10.0 lists the relevant references for this document. Chapter 11.0 contains the complete titles of papers, reports, symposia, talks, books related to the work under this contract. Chapter 12.0 gives the names of the researchers supported in part by this contract.

1.3 Principal Results

The major thrust of the work is divisible into five areas: complexity measures, test models and techniques, methods for developing programs with low error content, software reliability models, and software management models.

Complexity Measures

The approach to complexity measures has followed four avenues:

(1) An extensive study was made of recursive functions and their role for modeling program complexity. The results (reported in SMART 108-C

POLY EE/EP 77-037) show that despite considerable theoretical promise, recursive functions can only be applied to a narrow range of problems, and that with considerable difficulty. A component of the study, (reported in Sections 4.3 and 4.4 of the referenced report) involved the application of recursive functions to character strings. This work extended recursive function formalization (which is classically applied to integers), to the evaluation of character string elements. Ten operations on character strings are developed in the report. These allow an analytical description of nearly all important character string manipulations. This work should prove of value to researchers studying the properties of and operations on character string variables.

(2) A new complexity measure incorporating the program control structure was devised. This measure, named the polynomial complexity measure, allows the comparison of alternate designs and tells how to divide a program into modules for a minimal overall complexity. A technical report describing this measure forms volume 2 of this report.

(3) The research was continued on application of Zipf's law to program complexity. This work (initially reported in RADC-TR-4, Vol II, April 1978) provided a measure of program length from operator and operand counts. The present work expands on this theme. The previous effort measured just the bulk of a program, the newest effort takes the information content (i.e., the entrophy and thus the actual complexity, into account.

One difficulty in estimating the information content of messages - or in present context of programs - is that many probabilities are needed and yet the sample from which they are estimated is usually very small. Consequently, the present work is largely devoted to techniques and experiments in estimating entropies from small samples, that is, from small programs or from small sections of programs.

If the estimated entropy really corresponds to information content, one may expect the program to be invariant to translations such as assembly or compiling or writing it in a different style. Some success has been achieved along these lines, although an information measure simple enough to be of practical use cannot completely separate the "meaningful" information from the redundancies and "noise" code.

(4) An installation dependent, information theory based measure has been advanced. This measure has a number of desirable features:

- It includes all elements of a program, that may produce an error, even a simple assignment statement.
- It is more sensitive to infrequently used language features (and thus more error causing) than the commonly used ones.
- It weights heavier an element used in a complex manner than in a simpler manner. For example, a nested loop is perceived to be more complex than a sequence of two simple loops. Few of the existing measures have adequately addressed this situation.
- It allows for automated techniques in its calculation.

Test Models and Techniques

The research on test models and techniques consisted of three thrusts described below:

1. The first thrust was directed to the various aspects of verifying that a computer program correctly carries out the specified functions. The results (reported in RADC-TR-78-119, November 1978) allow one to determine the number of tests necessary to verify a program.

2. The second thrust was aimed at an actual implementation of an automatic software driver. In the classification of tests by types, type 1 test is one in which all flow chart paths are force-traversed at least once. Similarly, in a type 2 test, all flowchart paths are naturally-traversed at least once. The implemented driver is of type "1.5," that is one in which all paths are force-traversed, and some paths are naturally-traversed. The force-traversal is achieved by an algorithm for path analysis (that is, an algorithm for finding all possible paths) and the subsequent forcing of all deciders into accepting all their possible states with the resulting execution of all the paths.

This work (reported in RADC-TR-80-45) is a practical approach to testing, bypassing the much more difficult problem of input test data selection for the execution of all program paths.

3. The third thrust focused on statistical testing, constituting the volume 4 of this report. The application of this approach leads in some cases to an optimum testing strategy which minimizes the probability of failure.

Methods for Developing Programs with Low Error Content

The research on development of methods for designing programs with low error content focussed on obtaining quantitative comparison of programmer's peformance with two very different programming languages. One language was FORTRAN, and the other was the Klerer- May two-dimensional language. Programming in the two-dimensional language follows the familiar mathematical form and this minimizes the learning time (which consisted in fact of a single one-hour lecture) and requires a low degree of mathematical literacy. The results of the comparison indicate that even after a brief exposure, a programmer may construct an error-free Klerer-May two dimensional program in which less time than in FORTRAN. The obtained data shows the programming time rate of FORTRAN vs. Klerer-May as ranging from 1.76 to 7.1 for the performed experiments. A technical report describing this work constitutes volume 3 of this final report.

Software Reliability Models

The work on software reliability models concentrated on three areas:

(1) The analysis of software data and subsequent determination of

macro-model [1] parameters, specifically the E_T (i.e, the total number of errors in the program at the start of integration testing) and the K (i.e., constant of proportionality, relating the software failure to the number of remaining errors) parameters.

The estimation derived is a major improvement over the previous one. The old method involved moment or maximum likelihood estimation. An understanding of this method required a knowledge of statistics. The new method requires least-square-error fitting, which is simpler and allows graphical interpretation of changes in model parameters.

(2) In many practical situations the required data for performing software reliability estimates is incomplete. A method has been developed which provides bounds on the parameter estimates by imposing optimistic and pessimistic assumptions on the reconstruction of the missing data.

(3) A comparative study was made of the correlation between E_T and a number of problem complexity measures. Specifically, the measures compared were (1) expected instruction length of the resulting program, (2) operator and operand counts, (3) information content [2], (4) Halstead's effort measure [3], and (5) Akiyama's decision and call statement counts [4]. The highest correlation was achieved with Halstead's effort measure (with Akiyama's measure being a close second).

Software Management Models

The effort on software management models focused on two approaches. The first approach was applied to modeling of the development time and is described in [5]. The second approach studied the organization of a programming project and its effect on productivity. The initial ideas of the second approach were described in [5], and will be further detailed in Section 8.0.

1.4 Dissemination of Results

The results of the described research has been disseminated through the following channels

Published Reports	:	8
Published or Submitted Papers	:	15
Talks and Seminars	:	46
Books	:	4
Prizes & Awards	:	2
Organization of Conferences and Workshops	:	1

These are detailed in Section 11.0.

1.5 Major Accomplishments

Section 1.3 summarizes the principal results obtained. The major accomplishments were:

1. Halstead's pioneering work on software science was a major contribution to the field. Many investigators, however, have questioned the results because of the heuristic approaches in developing the key formulas. We have provided a basic theoretical frame work allowing one to derive Halstead's and additional formulas from fundamental principles involving Zipf's law, Shannon's theory of information and probabilistic models for program generation.

2. The major tool for removing errors and increasing reliability of software is program testing. The selection of test data to provide adequate coverage and exercise of the software under development is the major outstanding unsolved problem. The automatic test driver represents a significant contribution to testing. During the contract period previous experiments with drivers have matured into a practical implementation. Our driver, specifically, tests PL/I programs, and can be extended to other languages (such as, Pascal, Ada, JOVIAL, and so on).

3. A statistical theory of computer program testing has been developed. Even the most cautious and skilled tester will make mistakes of omission and commission. The theory incorporates the tester's performance by modeling the probability of his failing to recognize an erroneous result. The major result also accounts for the practical cases when (a) an input excites one or more errors or (b) one error requires multiple inputs for its discovery, and (c) when an input uncovers no errors.

4. A new measure of complexity has been developed. This measure, named the polynomial measure of complexity, takes the program structure into account. The measure models complexity by a polynomial rather than by a single number (as the popular cyclomatic measure does, for example). The measure addresses itself to (and solves) the important problem of the optimal decomposition of a program into modules, which minimize the overall complexity.

2.0 Introduction and Overview

2.1 State of the Art in Software Models

The last decade has brought about major progress in the field of software development. In essence this development has been transformed from individual art into a disciplined scientific approach. A large measure of the progress is attributable to the advances in software models and measures.

Reliability

There are nowadays several known and accepted measures of software reliability. These are Shooman's [6] exponential model, Jelinski-Moranda exponential model[7], Musa's time-history model[8], Littlewood's Bayesian model[9], Lipow's regression model [10], Shooman's-Natarajan error generation model[11], as well as Shooman's micro-model[12] (the last two models were developed with the support of RADC).

The above models as well as several others have been compared by several investigators. The most significant comparisons are the ones by A. Sukert [13], L. Duvall [14], and A. Schaeffer [15].

Availability

An availability model was developed for RADC by M.L. Shooman and A. Trivedi [16]. This approach uses a discrete-state continuous time Markov chain to model whether or not the software is in the up (i.e., operable) or the down (i.e., not operable) state.

Complexity

The difficult and important problem of assessing the complexity of a program has been investigated by many workers. A comprehensive summary of the work up to the close of 1979 was given by L. Belady [17] and B. Curtis [18,19].

RADC has supported the development of two significant complexity measures: the Zipf's law [20] and the polynomial measures [21]. Other important complexity measures were developed in the pioneering work on software science by the late Maurice Halstead [22].

Testing

The testing process has been modeled in the past by relatively few workers: Lipow [23], Mohanty [24], and Miller [25]. More recent work (supported by RADC) is the one by Popkin and Shooman [26], and Laemmel [27].

Management

It has been recognized for some time that management and organization structure of the project team play a vital role in the resulting quality, cost and schedule of software development. Brooks' book [28], which is based on the experiences encountered during the development of OS/360, focuses on these organizational problems. Unfortunately, the analytical work in this important areas is in its infancy. Recently models have been introduced (Tausworthe [29], Shooman [30], Cormier [31]) to describe the effects of the interactions of programmers with the organizational structure. The latter two models were developed with the support of RADC.

2.2 Need for Modeling Research

In an ideal situation ample proven models will exist for all aspects of program development, and these will be universally used by the practitioners in the field to predict and manage software project. In reality we have a Babel of models, ranging from excellent to useless, used by only a handful of practitioners. Most practitioners are reluctant to try the models because they are not convinced of their value. Researches point to several cases where the models have been validated, and cry for usage by and feed-

back from the practitioners, essential for further improvement. Thus, we have a vicious cycle which must be broken by cooperative researchers-practitioners ventures. A small step in this direction was undertaken under this contract through the vehicle of a major workshop in this area [32,33].

3.0 Applications of the Research to Software Engineering

3.1 Design Phase

An initial step in any design is to assess its scope, so as to marshall adequate resources for fulfilling the design objective. A rough approach is to compare the design with similar systems in the past and draw the conclusions. Such a comparison is inaccurate at best, and of little value in a typical military environment where each new task is much more demanding than its predecessor.

This contract supported work on complexity measures useful for gauging the difficulty of a design. One of these, the so-called "Zipf's law" measure extended the work begun under the previous contract to incorporate the actual entropy, that is, the information content, of the program into account.

The second measure, "polynomial complexity," incorporates the information of program structure as well as the extent of required testing. This measure recognizes the fact that the expected testing is a significant factor in the complexity of the design.

The contract also supported continued work on reliability models. It is as important to use such models in early as it is in later on operations. Shooman's macro-model predicts very early the expected number of errors and the duration of the integration test phase to meet a prescribed reliability requirement.

3.2 Testing and Debugging Phase

The research performed under the contract contributes to the testing and debugging phase with the work on: calculation of the number of tests for adequate verification of a program [26], automatic testing [34], and statistical testing [27].

The automatic testing technique focussed on the construction of a test driver. The driver forces the traversal through all the paths of the program by manipulating the conditions in the deciders. The statistical testing views the testing process as an incomplete one, and leads to cases for which an optimum testing strategy can be evolved and which minimizes the probability of failure.

3.3 Operational Phase

The macro-model described in the design phase is equally useful for the operational phase. Specifically, this model predicts the number of residual errors and thus tells whether or not the software is reliable enough to be released into field operations.

The work on the two-dimensional language [35] can be viewed as a contribution to the operational phase. The simplicity of the language and the high correlation with familiar expressions used in mathematics make it very attractive for maintenance. The details on this language are given in volume 3 of this report.

4.0 Complexity Measures

It is a fact of life that the estimation of important factors in software development leaves a lot to be desired. Underestimates by factors of two or of even ten in cost, manpower, computer time, development time and reliability, are commonplace. The villain is the elusive complexity, perceived by everyone, but in an ambiguous and fuzzy manner.

The earliest measure counted the lines of code, that is, the program bulk. A later popular measure, the cyclomatic measure, just counted the number of deciders, recognizing that deciders make a major contribution to program complexity.

The research under this contract attacked the complexity problem from several fronts. The output of this research are two completed studies, reported in technical reports, and two studies requiring a little more work before their culmination in technical reports.

The first attack focused on recursive functions and their application to program complexity. It has been conjectured for years that recursive functions should play a major role in describing a problem or even program complexity. The result of the study revealed the difficulty of applying recursive functions to practical problems, generally not limited to integer operations. Even though some progress was made in extending recursive functions to the domain of character strings, the conclusion is that we have to look somewhere else for practical tools for gaging complexity. For details we refer to the report [36].

The second completed study is named "The Polynomial Measure of Complexity," and constitutes volume 2 of this report. This study addressed itself to the fact that deciders and their structure are the main contributors to complexity. The role played by deciders has been recognized by McCabe [37] in his development of the cyclomatic measure. McCabe, however, just counts the deciders (for programs with no separate parts), ignoring the properties arising from their interconnections. One of these properties is the number of path tests necessary for a program of n deciders. For the minimal interconnection, it is just n+1, for the maximum connection it is 2^n, and for a third it may be any number between these two extremes. Clearly, the number of tests is a major cost factor requiring much effort, and is thus a major contributor to the program difficulty, that is, complexity.

To incorporate the structure of a program a description more general than just a number is needed. It is shown in the report that a polynomial describes a flowchart fully and uniquely, thus leading to the polynomial

measure. This measure is then applied to the key problem of optimal decomposition, that is, the decomposition of a program into modules whose overall complexity is minimal.

The two studies in progress are both directed toward exploitation of information theory in measuring program complexity. The first study expands the Zipf's Law measure from just measuring the program length to measuring the actual information content, that is, the program entropy. It does so by sampling small program segments and calculating eutropy from actual frequencies of commands in several languages. Present work experimented with English, IBM/370 assembler, PDP-11 assembler and PL/I to obtain a feel for the problem, and the underlying language redundancies. The object of immediate research is to estimate the entropy from a sample whose length will ordinarily be too small, because in a typical short program there is rarely a sufficient number of small occurrences to perform the calculations adequately. The method used is the "jack-knife" one. This method consists of subdividing the sample into finer and finer divisions (each division being a power of 2) thus generating a small enough sample and then taking the limit as the sample length approaches infinity. The details are left to a future technical report.

The complexity measure of the second study in progress has several goals. As a first goal it is based on all elements of the program; it recognizes that everything can produce an error, even a simple assignment or move instruction.

As the second goal, it wants the measure to be more sensitive to infrequently used language elements than to the commonly used ones. This aspect recognizes the intuitive reality that one does more accurately that what one does frequently.

The third goal is to model the measure so that an element used in a complex manner contributes more to the complexity than the same element used in a simpler manner. For example, a nested loop is intuitively more complex than a sequence of two simple loops. None of the measures advanced so far have adequately addressed this situation.

The final goal is the allowance for automating of the calculation. This is essential, because this measure is a rather extensive one, thus making a hand calculation completely unattractive.

5.0 Test Models and Techniques

The important topic of test models and techniques has not received as yet the attention it deserves in the published literature for a simple reason: it is a very difficult nut to crack. We have completed three studies (and published the technical reports), two of which are continuing.

The first study [26] considered the problem of determination of the number of tests needed to test a program. For details, the reader is referred to the published technical report.

The second study considered the construction of an automatic test driver [34]. This driver forces all deciders into their all possible states, and thus provides a traversal through all possible paths. Again, the details are given in the published report.

The third study culminated in a statistical theory of program testing. It addresses itself to the problem of developing an optimum test strategy in the light of the fact that the test, even if passed, is a partial one at best. Again, the details appear in the published report [27].

The first and the second studies are continuing. In the continuation of the first study, the object is to obtain an improved bound on the number of tests, and to do so with less computational effort. In particular, the ways of decomposing a flowchart are investigated with the goal of finding the best method of decomposition (best, in the sense of leading to the most accurate bound from the bounds of the individual partitions).

The continuation of the second study addresses itself to the development of a more general driver, and with fewer flaws. The present driver tests even paths that could never be reached by natural execution, thus creating error messages for non-existent errors. To remove this flaw we need a driver of higher type (higher than type 1), because unreachable paths cannot be determined by a static analysis. Another restriction is that in the present driver certain legal PL/I constructs (e.g., multiple closures) cannot be handled. Also, if further support for this work is made available we will write a test driver for programs written in Ada.

6.0 Program Methology for Low Error Content

Because of the difficulty in producing software with low error content, we constantly look for a better mousetrap, that is, for a better language or method that will avoid or reduce the errors inherent in released programs.

We reported earlier [38] on the work on automatic programming. This work was dormant during the two year span of the present contract. Instead, the approach was to investigate the merits of a language closely following the familiar format. The experiments with such a language, the Klerer-May two-dimensional language, and FORTRAN were undertaken. The details of the experiment, as well as the results obtained are reported in volume 3 of this report.

7.0 Software Reliability Models

The principal thrust of the present effort was directed toward making the model useful for the practitioner in the field rather than just for the researcher. The macro model [1] was subjected to two-pronged attack. First, the model was validated with field data to insure its effectiveness in prediction. This proved to be successful, and consequently a method was sought to simplify the estimation of model parameters. The existing methods (of moment or maximum likelihood estimation) are not in the set of tools of an average practitioner, thus inhibiting significantly the model use.

The simplified technique uses only straight-line least-square error fitting [32]. The least-square error fit has the following advantages:

1. Least square error fitting is well known and understood by the technical community.
2. Approximate straight-line fitting by eye is very quick and approximates well a least-square error fit.
3. Every programmable calculator or computer contains built-in least-square fitting software.
4. Since a picture is worth 1,000 words, least squares with it pictorial display provides insight, confidence, and trends.
5. The two model parameters are the intercept and the slope. The graphical display portrays the sensitivity of these parameters to data errors (or differences in interpretation).

A further effort on increasing the usefulness of models focussed on the common case when there is not enough data to estimate model parameters. The result of this effort is a technique for obtaining a range for model parameters.

8.0 Software Management Models

Two approaches to management models were undertaken. The first approach modeled the development time and is described in [5]. The second approach analyzed the effect of organizational structure on the software management and will be described here.

8.1 Introduction

Software engineering encomposes all activities required for the planning, design, development, generation, maintenance, enhancement and modification of software. Since the most critical resource in software engineering is creative human thought, people management is of paramount importance. The importance of the management function on software engineering projects has been recognized repeatedly in the literature. One survey done in 1975 on the Safeguard Data Processing System concluded that "the shortage of experienced software managers on the project posed a more serious challenge than the shortage of experienced programmers." [1]

Since an organization's management policy is implemented in its organizational structure it follows that the particular organizational structure selected for the management of a software project will contribute significantly to the degree of sucess achieved. This effect was recognized in a report on

---

[1] J.D. Musa and F.N. Woomer, Jr., "Safeguard Data Processing System: Software Project Management," Bell System Technical Journal, 1975, pp. S245-S259.

software modelling studies produced in 1977, which came to the conclusion that "...the organizational structure of the project team has a large influence on the productivity, reliability and quality of the software produced."[1] An article in the December, 79 issue of Datamation asserted the fact more forcefully: "Software managers who succeed in establishing effective organizations will enjoy development rates 1,200% better than managers who fail."[2]

If the effect that the organizational structure has on software engineering can be quantified, it will be possible to select the optimum structure for any given project. A quantitative model has been developed which utilizes graphs of the organizational flow of control and information. Values are assigned to each unit of information transmitted (α) and each link formed (β). The model is constrained by the maximum number of interfaces that an individual can handle effectively. Optimization of the organizational structure is achieved by minimizing the result of trade-offs between the α's and the β's.

8.2 Software Engineering Management

In most software projects development costs are the total system costs, since the prototype software system is the only one produced. That is in sharp contrast with hardware projects where large-scale production constitutes a major portion of system cost. Since the most critical (and expensive) resource in software development is creative human thought, people-management takes on increased significance. It becomes vital that the optimum number of mix of designers, coders and testers be assigned to the project; that they be organized into the optimum structure; and that they be provided with the required management tools for controlling development efforts and documenting results.

Thus, management of software engineering consists of managing human creativity so as to maximize productivity. This necessitates identifying what productivity is. Brooks [28, pp. 83-90] has shown that it is not necessarily proportional to charged man-hours. We submit that productivity is primarily a product of managment policy as implemented in organizational structure. We further submit that what is implemented in these key areas is a manifestation of management philosophy. The relationship among philosophy, policy and organizational structure is depicted in Figure 1.

8.3 Management Policy and Organizational Structure

A supportive management policy is implemented chiefly through an organization which provides managers with authority commensurate with their

---

1 M.L. Shooman and H. Ruston, "Final Report: Software Modelling Studies," Program in Software Engineering Poly-EE-77-042, SRS112, Sept. 30, 1977 pp. 23 and 24.

2 Edmund B. Daly, "Organizing for successful software development," Datamation, December 1979, pp. 107 to 120.

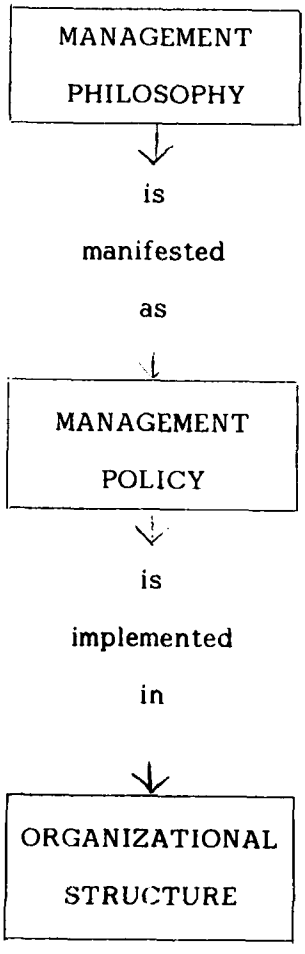


FIGURE 1

RELATIONSHIP AMONG MANAGEMENT PHILOSOPHY, POLICY AND ORGANIZATIONAL STRUCTURE

responsibilities, and which promotes and rewards free exchange of information between individuals, sections and departments.

Organizational structure determines the flow of control - who controls which resources (human and others), and who has to account for which results. It also determines the flow of information - who has access to what, and what information is collected.

There are practically as many different organizational structures as there are managers. However, three distinct types (project, functional, and matrix)[1] emerge, and other structures can be considered to be a composite of the basic types. These are best illustrated by an example. Let us assume a new development organization is required to develop two projects: Project A and Project B. Each project has three major functions to perform: real-time software development (operating systems), support software development (compilers), and hardware development (computers).

Figure 2 shows six separate organizational entities, one entity for each technology for each project. Now the manner in which we combine these separate organizations will give us a project organization structure, a functional organization structure, or a matrix organization structure.

In a project organization structure (Figure 3) all resources required to complete a project are organized under a single line manager who performs both the technical and administrative functions. A functional organization (Figure 4) groups all the people associated with one speciality under a functional manager (e.g., all real-time software development for all projects). A matrix organization (Figure 5) attempts to incorporate the advantages of the other two basic structures, project and functional. Personnel are grouped functionally for technical and administrative purposes, but are responsive to a project manager. The project manager decides what will be done, while the functional manager decides how to do the job, and supplies all resources.

The organizational structures that will be examined using the quantitative graph model will be made up of elements of the above three basic types in varying degrees, and therefore will be advantageous or disadvantageous depending on the value/cost/effort associated with the model parameters. The parameter values, in turn, will be determined by the management exigencies of a given project.

---

1 Daly, op. cit. p. 10

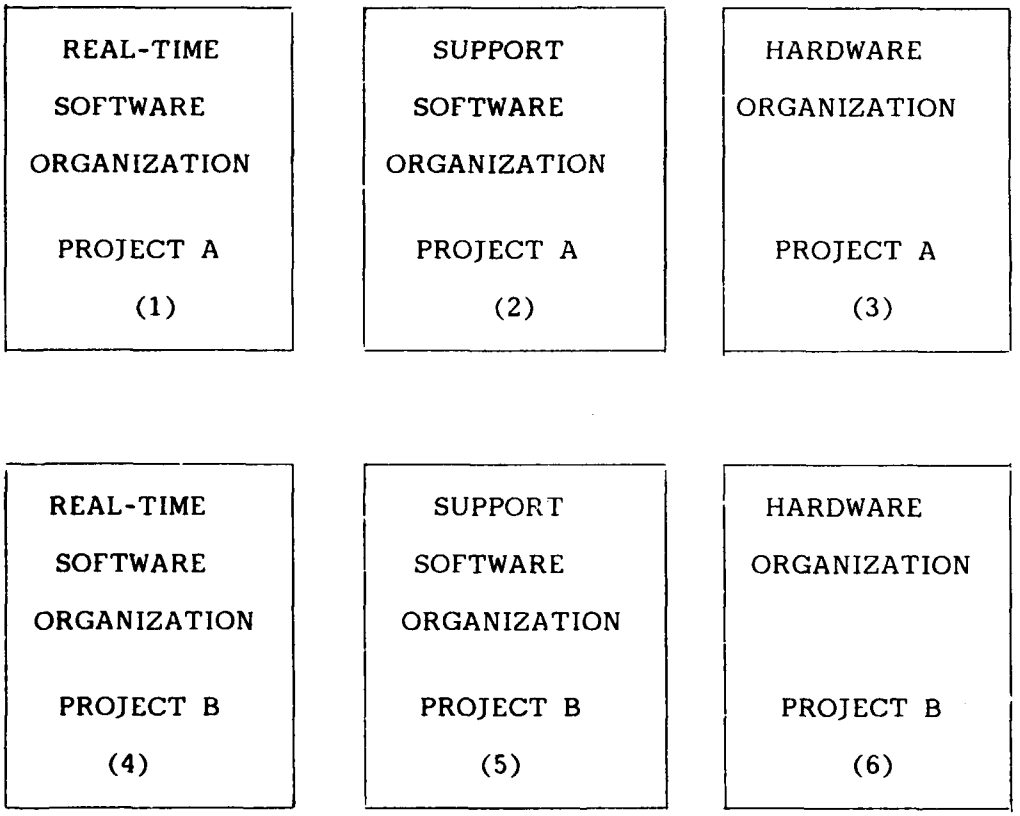


FIGURE 2

SIX ORGANIZATIONAL ENTITIES

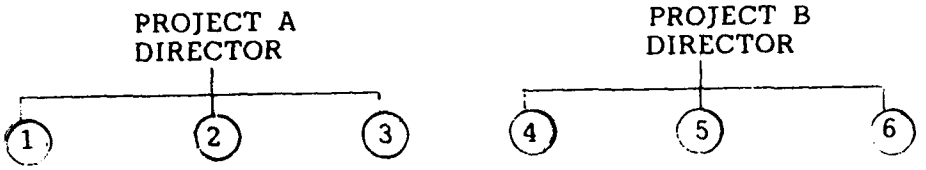


FIGURE 3
PROJECT ORGANIZATION

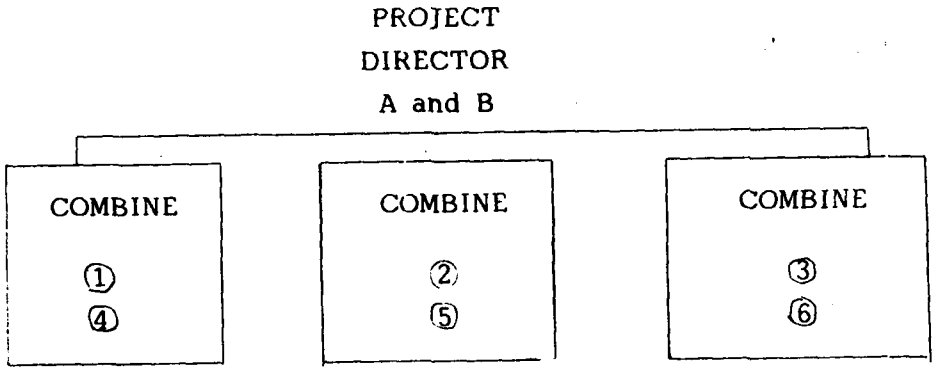


FIGURE 4
FUNCTIONAL ORGANIZATION

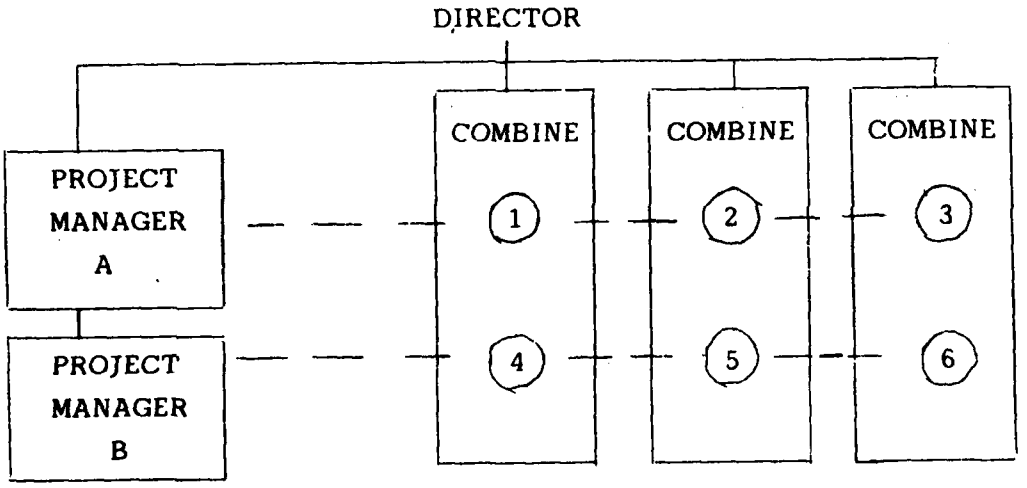


FIGURE 5
MATRIX ORGANIZATION

8.4 Productivity Models

Most quantitative models which examine the effect of organizational structure on productivity assume a "worst case" situation. Shooman [2] observes that software development productivity is not a direct function of charged time. Charged time represents raw man hours composed of personal time (coffee breaks, conversations, etc.), communication time, and lastly, productive time. He assumes that the proportion of personal time is fixed at 10% regardless of the organizational structure. However, the remaining time divisions are highly dependent on the organizational structure. He proposes a model which breaks the total time (T) into development time (T_d), and communication time (T_c).

$$T = T_d + T_c \tag{8.1}$$

He then postulates that for a project team consisting of N_d workers, every team member communicates with every other team member. Thus the number of interfaces is the number of combinations of N_d taken two at a time.

$$\frac{N_d}{2} = \frac{N_d!}{2!\ N_d!} = \frac{N_d\ (N_d - 1)}{2} \tag{8.2}$$

If there are L total lines of code to be developed and T_d is measured in months, then the productivity (P) in lines/month is given by

$$P = \frac{L}{N_d\ T_d} \tag{8.3}$$

If we assume that a certain fraction (K) of the total work time (T) is spent communication with each interface, then T_c the total communication time for all the interfaces is given by

$$T_c = KT\ N_d\ (N_d - 1)/2 \tag{8.4}$$

Substituting equation (8.4) in equation (8.1) and solving for T yields

$$T = \frac{T_d}{1 - KN_d\ (N_d - 1)/2} \tag{8.5}$$

Multiplying both sides of equation (8.5) by N_d and substituting from equation (8.3) yields

$$N_dT = \frac{L/P}{1 - KN_d\ (N_d - 1)/2} \tag{8.6}$$

The numerator in equation (8.6) predicts a linear variation in man months with program length; however, the denominator factor produces a plot which curves upward indicating a decrease in productivity due to the increase in N_d for larger programs.

Similarly, Tausworthe [29, Chapter 10] measures software team productivity by defining "index of productivity" (P) in terms of the total number of lines of code (L), number of workers on the project (W), and the average time each worker spent developing the software (T) by the formula

$$P = \frac{L}{WT} \tag{8.7}$$

T is then split into productive time (T_p) and non-productive time (T_{np}) spent interfacing with each of the other team members.

$$T = T_p + (W-1)T_{np} \tag{8.8}$$

He then postulates the individual productivity level (P_i) that each team member must sustain during his "productive" time periods so that the team have overall productivity P is given by

$$P_i = L = \frac{WP}{1 - (W - 1)(T_{np}/T)} \tag{8.9}$$

After extensive formula manipulation he arrives at the conclusion that the amount of code that a project can produce per day has a maximum value, found to be

$$\left(\frac{L}{T}\right)_{max} = P_i \frac{1 + (T_{np}/T)}{2\,(T_{np}/T} \quad \left[\frac{1 + (T_{np}/T)}{2}\right] \tag{8.10}$$

where the figure in braces represents the loss in personnel efficiency. This maximum production rate is achieved when the team size is

$$W = \frac{1 + (T_{np}/T)}{2\,(T_{np}/T} \tag{8.11}$$

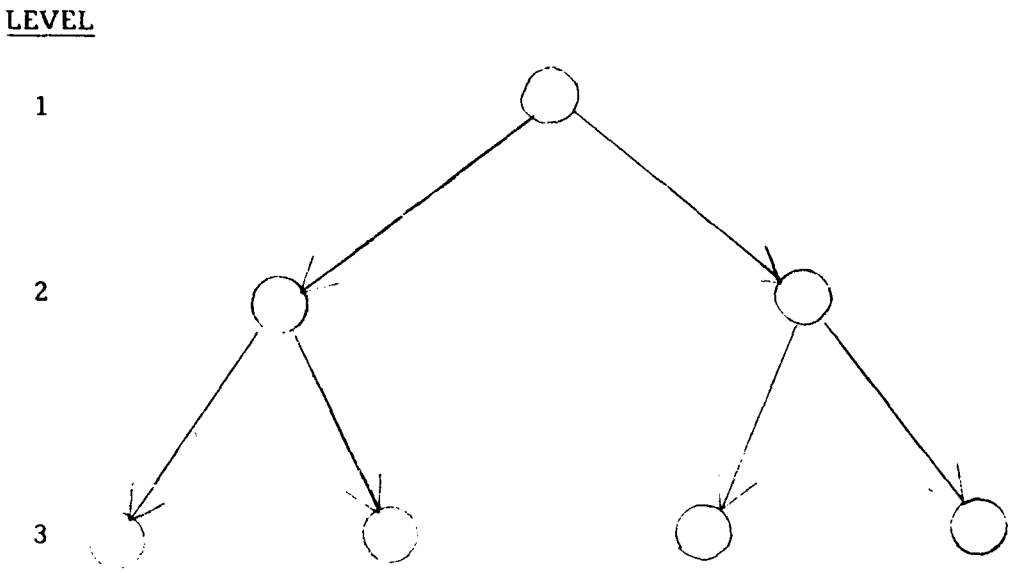
He then concludes that a project hoping to deliver L lines within time T using W workers having individual integrated-task productivities P_i must keep their nonproductive index (T_{np}/T) within the bound

$$\frac{T_{np}}{T} < \frac{1 - (L/WTP_i)}{W - 1} \tag{8.12}$$

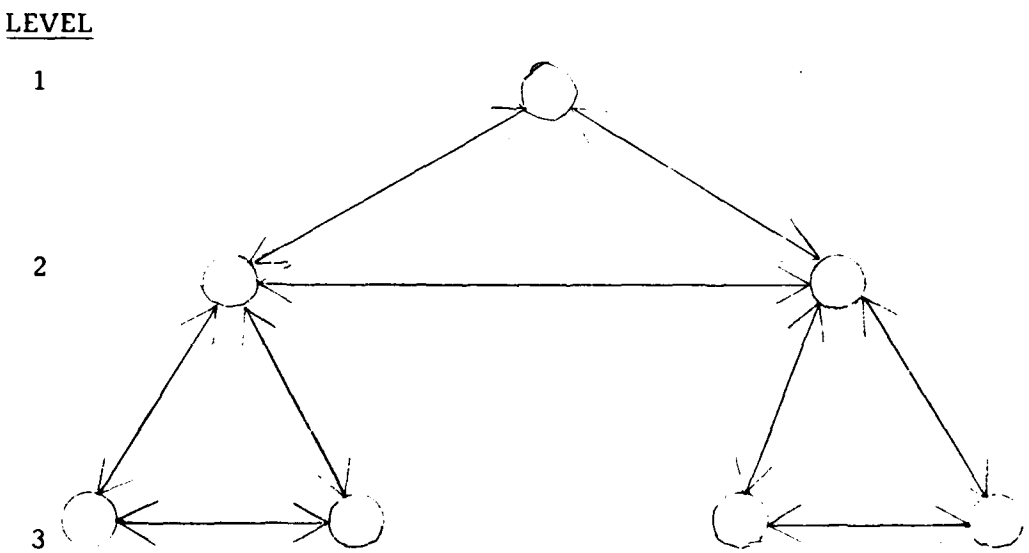
if there is to be success.

8.5 Quantitative Graph Model

The preceeding two models assume that each team member interacts with every other team member. What if the team is organized into a different structure? A quantitative model has been developed [39] which utilizes graphs of the organizational flow of control and flow of information structures to evaluate the effect of alternate organizational structures on the number of communication paths, and on the volume of information flowing through each path. The concept is illustrated in Figure 6. The lateral paths in the



(a) FLOW OF CONTROL



(b) FLOW OF INFORMATION
FIGURE 6
QUANTITATIVE GRAPH MODEL

information flow model are analogous to the pre-review discussions implemented in the "Generic Engineer" concept, [40] and are considered to be essential to effective monitoring of progress. The models can easily be extended or transported to desired level of detail.

8.6 Graph Model Parameters

To illustrate the graph parameters we will decompose a problem, P, into three subproblems - P1, P2, P3 - of identical size. Thus instead of solving P, we can solve P1, P2 and P3. The three subproblems are in general related to one another, i.e., some effort must be expended in having them communicate with one another or coordinate them. (This may take the form of engineers spending time coordinating their proposed solutions to software subproblems). We wish to study the effect of the shape of the system on its overall cost or complexity. Two alternative organizations will be compared. In case 1, (Figure 7 (a)) each person communicates directly with every other person. In case 2 (Figure 7 (b)) all communication is routed through a central point (P2).

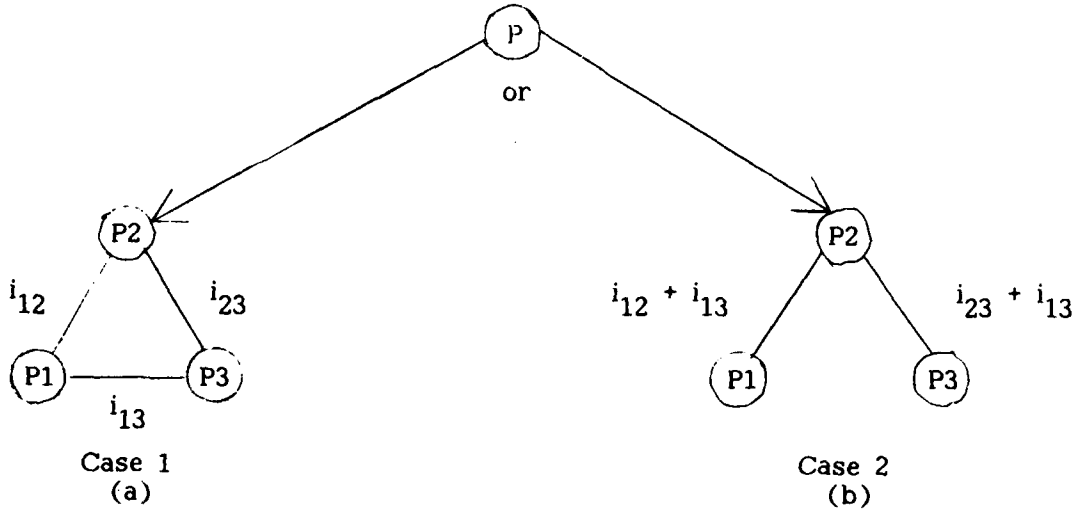


FIGURE 7
DECOMPOSITION OF A PROBLEM P INTO THREE SUBPROBLEMS

Let i_{ab} be the amount of information exchange required between persons a and b in order to complete their tasks. We define a constant, α as the amount of effort per unit of information exchanges, i.e., the total effort required to exchange i_{ab} units of information is αi_{ab}.

We define another constant, β, as the amount of effort associated with the existence of a direct communication path between any pair of persons. This corresponds to the overhead cost of establishing direct communication which is independent of the amount of information exchanged (e.g., the cost of having a meeting, not counting the time spent actually exchanging information). For problem P, letting $i_{ab} = I$ for all a b, (i.e., constant information exchange) it can be shown that direct communication (Figure 7 (a)) is more efficient if $\beta < \alpha I$, and the use of an intermediary (Figure 7 (b)) is better for $\alpha I < \beta$. Generalizing, in the case where the i_{ab}'s are different, the use of an intermediary is efficient if

$$\beta > \alpha i_{ab}$$

and there is some c such that direct interfaces between (a and c) and (b and c) exist.

It is apparent that the value assigned to the model parameters, α and β will determine the optimum structure. This is realistic since the importance given to accuracy of information and interpersonal interaction will vary from one project to another.

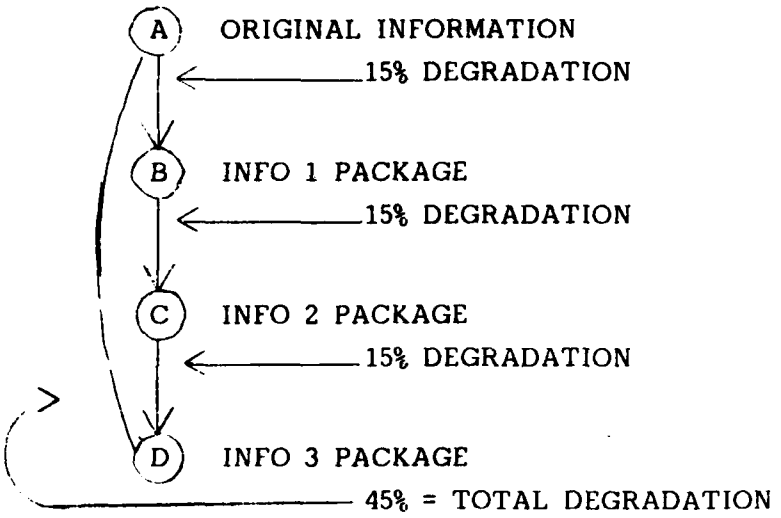
A gross analysis of the model as developed to this point reveals that:

(i) if αi_{ab} is small relative to β -implying either that very little effort is required to transmit one unit of information over one link (α is small), or that very few units of information are required to be exchanged between a and b (i_{ab} is small) - then the use of intermediaries is favored.

(ii) if αi_{ab} is large relative to β - implying that the overhead cost of a direct link is small compared to the cost of exchanging information - then the use of direct communication is favored.

8.7 Refinement of the Parameters

Software engineering managment information is transmitted verbally, in writing, and through transcription. The accepted retention rate for information transmitted verbally is between 40% and 60%. The integrity of information transmitted in writing and via transcription is higher. A realistic average degradation of information content for any one transfer of information will therefore be taken to be 15%.



TRANSMISSION-CAUSED DEGRADATION OF SOFTWARE ENGINEERING MANAGEMENT INFORMATION

FIGURE 8

We illustrate the concept of information degradation in Figure 8. An original information package is to be exchanged successively among four individuals (A,B,C and D). In being transmitted from A to B the original information content suffers a 15% degradation yielding the new information package, info 1. Info 1 in turn is degraded by 15% in the transmission from B to C yielding info 2. Info 2, when transmitted from C to D, is similarly degraded yielding info 3. If D were to communicate the info 3 package back to A, we would find that little more than half the original information content remains.

Our graph model takes into account information degradation by incorporating it into α, the cost of transmitting one unit of information over one link. The degradation cost is essentially the cost of checking for and correcting errors. Thus, information degradation is directly proportional to the number of information units transmitted, and to the number of links over which the information is propagated.

8.8 Capacity Constraints

There is a limit to the number of interfaces that one individual can handle. (Were this not so, then the lower boundary on the number of links required would be represented by the case in which there exists only one intermediary through which everyone communicates.) This capacity constraint has been studied extensively for organizational flow of control. Theories and techniques for optimizing an individual's span of control abound.

In an organization where the work is simple, routine, and repetitive - like the basic kind of assembly work - a supervisor might be able to handle 25 to 30 people and do all the necessary supervisory work. If, however, the work managed is variable, the supervisor must spend more time to set objectives, to train, to put in new methods, and consequently cannot handle as many people.

In the realm of software engineering management is quite a complex function; therefore, an individual supervisor can supervise only a limited number of people. Schleh [41] attacks some traditional methods of "spanning the gap." He refers specifically to the tendency among companies to feel that they have so many supervisors at the first level that they could easily cut out one or two, have each remaining one handle a little more, and still get by. This is an illusion. One large paper plant did this and found, within three years, that its cost increased 15 percent, and quality slipped. Costs and quality improved only after the span of control for each foreman was decreased and each could handle his work.

On the other hand, executives often fail to grasp that many of their communication problems come from too long a management chain. If the first supervisory level is not beyond its span of control, the second and third can handle many more managers. In one plant a superintendent supervised four foremen. When the foremen were set up with more manageable span of control and trained to supervise, a superintendent could supervise eight to nine foremen. Communication problems up and down the line were greatly decreased because problems were solved in most cases by the foremen.

When applied to organizational flow of control and organizational flow of information graphs, capacity constraints may cause an otherwise optimal solution to become unfeasible. For example, the lower bound solution of making one person an intermediary for all others, generally optimal for $\beta >> \alpha$ may be unfeasible as it places a tremendous burden on the intermediary and may violate his capacity constraints. In such a situation, (not common in the functional organization structure), it may be necessary to introduce additional personnel strictly as intermediaries in order to satisfy the capacity contraints. For a given person (a), if we let his total capacity equal C_a and the capacity required to solve the subproblem assigned to him equal to R_a, then his spare capacity available for communication (S_a) is equal to:

$$S_a = C_a - R_a$$

In order for a feasible solution to exist

$$S_a \geq (\Sigma_b \alpha i_{ab}) + \beta$$

i.e., each person must have at least enough spare capacity to handle his own communication requirements plus one interface to someone else.

This work has been applied to three basic organizations structures portrayed in Figures 3, 4, and 5 (project, functional, and matrix). The following conclusions were reached.

a. The effect of organizational structure can be quantified
b. An organization structured to provide lateral exchange of information maximizes productivity.

Further work is needed in this vital area. Two promising avenues for research are:

a. The resulting graph model when α and β parameters are nonlinear.
b. A comparison of Shooman's and Tausworthe's productivity models.

9.0 Other Research in Progress

9.1 Introduction

As in any research endeavor, the different tasks are in various stages of completion. Some have reached a point which justified comprehensive research report (enumerated in Section 11.2). Other tasks will be described here in order to document the progress to date.

The eleven technical sections of this chapter can be broadly classified into the following categories:

1. Complexity: 9.2, 9.3, 9.4, 9.5
2. Models: 9.6, 9.7
3. Further Results on the Applications of Zipf's Law 9.8
4. Cost Estimation 9.9
5. Programming Methods for Low Error Contents 9.10
6. Testing Methodology 9.11

Each of these sections is self-contained with appropriate literature references to related work. It is anticipated that several of these will be developed in the future into comprehensive technical reports or research papers.

9.2 Psychological Complexity.

This perception of complexity relates to the understanding of a program. Up to now this work has not been related to the numerical complexity arising from properties of the control graph. The studied properties of the control graph, such as the number of regions (i.e., the cyclomatic complexity) or the number of intersections (i.e., the knot count) or the properties of length (i.e., maximum or mean path length), width (the number of parallel paths), or area (the product of length and width) have not been correlated with psychological complexity. Initial work in this area has explored the correlation between expert judgement of a complexity of program with the above quantitative geometric complexity measures. This was done on a small scale with student-type badly-documented, programs. Further work should ascertain correlation for practical program (with the usual amount of annotative material) and relate it to the experiment conducted by Amster [42].

9.3 The Information Theory Approach to Complexity

This approach treats a number of residual errors as a manifestation of complexity. It assumes that less commonly used language features, both syntactical and structural are less familiar, and thus give rise to more errors than the familiar constructs. This conjecture has been tested by automatically counting the frequencies of various language features and correlating with the number of errors associated with this feature. The present data is incomplete and further data gathering and analysis are required.

9.4 The Polynomial Measure of Complexity

This measure is described in detail in volume 2 of this report. There is still more work to be done on this interesting measure. Firstly, the present comparison of complexities of two polynomials $p_1(x)$ and $p_2(x)$ is either based on both the coefficients of the corresponding powers and the testing efforts, these being $p_1(2)$ for $p_1(x)$ and $p_2(2)$ for $p_2(x)$. The question of comparison of polynomials of unequal degrees is still largely an open one. To this and other such issues one needs a testing effort to relate this measure to the other geometrical and subjective (i.e., psychological) measures.

9.5 Application of Theory of Hardware Complexity to Software Complexity

The object of this research is to apply the well studied theory of hardware complexity to problems involving software complexity.

Hardware complexity is well understood at the gate level through the synthesis procedures of Boolean functions. We know how to minimize switching circuits, we know how to build in redundancies for reliability, and we know how to exchange and manipulate equivalent structures.

A programming process manipulates input data into output data in several stages. In each stage some or all of the previous variables are modified. This is analogous to a switching function applied to signals in a switching interval. We have attempted to view the programming process as a switching process and derived trade offs between processing time and memory. Complexity was considered as the number of gates, number of used memory cells (1-bit cells), and the needed execution time. The result was a hyperbolic cylinder in a 3-dimensional region in the space of gates, memory cells, and execution time.

9.6 Software Reliability Data Analysis and Model Fitting - A Case History

A set of error data gathered from field trials of a medium-sized real-time computer system has been analyzed. The software was 50K in size, and ran on a microcomputer which performed measurements on a batch of

samples and issued reports. The reliability model used assumed that the failure rate was constant and proportional to the number of remaining errors[43,44,45]. The two model constants, K´ the proportionality constant, and E_T the total number of errors, were determined by two different methods. During the three-month field trials 176 tests hours were accumulated, 132 errors were removed, and the system exhibited a mean time between failure (MTBF) of about three hours during the third month. The model predicts: (1) there are about 200 total errors, and (2) about 100-200 additional hours of testing are needed to raise the MTBF to 100 hours. This goal appears to be unrealistic. The following tells how hardware, software, and operation failure rates interact to yield system reliability, and discusses how MTBF goals should be chosen.

This work is based upon actual experiences in predicting the reliability of software. The 80 errors which are discussed were found by investigating system failures which occurred during field trials of the system. These field trials were held as part of the software integration phase of the development process rather than subsequent to the integration phase as is generally the case.

The objectives of this work were:

1. To illustrate the use of a model[43] in predicting software reliability from failure data.

2. To investigate how to overcome the uncertainties of incomplete or erroneous data and to evaluate the accuracy of the resulting estimates.

3. To compare different model fitting methods and the sensitivities of the resulting parameters.

4. To study how accurate the software reliability estimates must be for management decisions.

5. To explore how one should establish mean-time-between-failure (MTBF) requirements (or goals).

Some of the characteristics of the program and the environment in which it was developed are listed below:

1. The program was basically a process controller, performing measurements in real-time on a batch of samples, and producing reports as output.

2. The developer was a division of a large (FORTUNE 500) company.

3. This was their first large software project.

4. The host computer was a modern minicomputer.

5. Many portions of the software were written by outside consultants, but the remainder and the system integration phase were done by in-house designers.

6. The programs in question totaled about 50K lines of object (machine) code. The programmer wrote (source code) in a mixture of assembly language and higher level language.

7. No formal configuration control or error recording techniques were used.

8. All data on operation and errors which occurred during field trials were kept in narrative form in a system operator's notebook.

Summary of the Reliability Model. The reliability model used in this paper has been described in detail in a number of references[43,44,45]. In brief, the model assumes that the program enters the integration test phase with E_T total errors remaining in the software. As integration testing proceeds, all detected errors are promptly corrected, and at any point in the development cycle (after τ months of development time[1]), a total of $E_c(\tau)$ errors have been corrected, and the remaining number of errors is

$$E_r(\tau) = E_T - E_c(\tau) \tag{9.1}$$

The above is often normalized through division by the number of object code instructions, I_T

$$\varepsilon_r(\tau) = \frac{E_T}{I_T} - \varepsilon_c(\tau) \tag{9.2}$$

where

$$\varepsilon_r = e_r/I_T \qquad \varepsilon_c = E_c/I_T$$

If we assume that the failure rate, z(t), is proportional to the number of remaining errors, then

$$z(t) = C\ \varepsilon_r(\tau) = \frac{C}{I_T} E_r(\tau) = K' E_r(\tau) \tag{9.3}$$

where C is a normalized proportionality constant and K´ is a constant for the given program length I_T.

---

1 In this paper the actual number of test hours is estimated and is used as the development time variable rather than the cruder calendar days.

Using the principles of reliability theory[46], we obtain expressions for the reliability function, R(t), and the mean time between failures (MTBF):

$$R(t) = \exp[-K'E_r(\tau)t] = \exp[-K'(E_T-E_c(\tau))t] \quad (9.4)$$

$$MTBF = \frac{1}{K'E_r(\tau)} = \frac{1}{K'[E_T - E_c(\tau)]} \quad (9.5)$$

(In this project it is difficult to accurately estimate I_T for several reasons. Thus we have chosen to use the model including the constant K'.)

Data Analysis. Generally, the most difficult task in reliability estimation is the analysis of failure data, and such is the case with this project. The biggest problem is that the software was not under configuration control. Thus, we do not know exactly how many errors were found in the software, nor the date at which they were removed. By carefully interviewing one of the principal designers of the software (who was in charge of the field trials), we were able to identify all the field software errors and determine whether or not they reoccurred later during the trials. However, while the field trials were in progress, the rest of the in-house development team was testing and debugging the software, and we were unable to obtain a record of these errors (if one existed).

During the three-month segment of the field trials analyzed, 81 different software errors occurred one or more times, for a total of 132 software error occurrences. Also, six hardware errors and one operator error were recorded. The monthly totals are shown in Table 1. Note that although we don't know when an error is corrected, we do at least have a lower bound on the removal time for errors which reoccurred. Thus, the meaning of seven errors carried over from the month of March to April means that these seven errors have reoccurred in April or later. Consequently, they could not have been removed during the month of March. The number of working days excludes holidays, days when the field trial was not manned because the operator was back in the development lab, and interruptions for installation of hardware modification and hardware repairs. The time estimate of four hours of test time per day is a rough guess, since no hours were recorded.

An initial set of software reliability computations is made in Table 2 along with estimates of the number of errors removed, based upon various assumptions. (These error removal estimates are further discussed in the next section.) Before elaboration of this data, however, we must examine how the reliability goals were chosen for the system.

The reliability goal set for the entire system was 500 hours MTBF, although the rationale behind this choice was never fully investigated. The failure rates of the hardware, software, and operator must all be included in a prediction of system reliability. Unfortunately, the initial thinking excluded the software and the operator. An estimate was made of the computer hardware reliability, and after a few iterations with the minicomputer

Table 1
Error Occurrence During the Field Trial Period

Time Period 1979	Distinct Software Errors Found	Un-removed Errors	Error Occur-rences	Hardware Errors	Operator Errors	Test Days	Test Hours
March (19-30)	32	0	41	3	1	6	24
April (1-30)	31	7	58	2	0	16	64
May (1-31)	14	3	23	1	0	14	56
June (1-13)	4	3	10	0	0	8	32
Totals (March 19-June 13)	81	-	132	6	1	44	176

Table 2
Reliability Computations

Time Period 1979	Test Hours	Software Failures	Failure Rate z	MTBF	Cumulative Number of Errors Removed at Beginning of Period		
					Assump-tion1	Assump-tion2	Assump-tion3
March (19-30)	24	41	$1.71hr.^{-1}$	0.59hr.	0	25	50
April (1-30)	64	58	0.91	1.10	25	60	120
May (1-31)	56	23	0.41	2.43	60	74	148
June (1-13)	32	10	0.31	3.20	74	81	162

manufacturer's reliability group, a MTBF estimate of 630 hours for the computer hardware emerged. Since this seemed to predict a modest safety factor, the matter was closed until much later when questions about the software failure rate were raised.

How should the system reliability goals have been set initially? We suggest that because of the lack of any other relevant information on the system, it is reasonable to require that the MTBF of the software be equal to the MTBF of the hardware. In addition, we might require that the operator MTBF* be five times greater than the hardware and software; in this case 2500 hours. It is easy to show that if hardware, software, and operator failures have a constant failure rate (exponential density function), and if all failures are system failures, then we can write: **

$$R(t) = \exp[-(\lambda_s + \lambda_h + \lambda_o)t] \tag{9.6}$$

$$MTBF = \frac{1}{\lambda_s + \lambda_h + \lambda_o} = \frac{1}{\frac{1}{MTBF_s} + \frac{1}{MTBF_h} + \frac{1}{MTBF_o}} \tag{9.7}$$

where

λ_s, λ_h, λ_o = the failure rates of software, hardware, and operator, respectively. $MTBF_s$, $MTBF_h$, $MTBF_o$ are the mean times between failures of software, hardware, and operator, respectively. Subsitution of MTBF values of 500, 500, and 2500 into Eq. (9.7) yields a system MTBF of 227 hours. Based on the previously described hardware prediction, the 500 hr. goal for the hardware seems to be realistic and reachable. To be on the safe side, if we really hope to meet the 500 hr. and 2500 hr. goals for the software and operator, we should design toward slightly higher goals to leave a little safety margin to cushion us against estimation errors, unknown contingencies, bad luck, etc. One should then investigate whether the software and reliability goals are realistic by making predictions, obtaining data on predecessor or similar systems. This issue will be discussed further in the concluding section.

We now return to our evaluation of the data gathered during the field trials. Since we have amassed a total of 176 hours of operation during which six hardware and one operator failure occurred, (c.f. Table 1), we compute the hardware and operator MTBF as 176/6 = 29 hr. and 176/1 = 176 hr., which are far below our goals. Since several of the hardware failures were related to memory disk and interface problems, which were subsequently fixed, one would anticipate that more recent tests of the hardware will show improvement. The operator errors suggests effort should be directed toward a good operator's manual and operator training.

---

* Since the operator manually loads the batch of samples and chooses the mode of system operation, there is ample opportunity for an occasional human error.

** See Ref. 46, p.221.

Similarly, we compute software MTBF for each month. In Table 2 we see that the software MTBF is 0.6 hr. during the month of March and rises to 3.2 hours in June. The growth is graphically depicted in Fig. 9.

We reach the inescapable conclusion that both the hardware and software are still in development test stage, and considerably more work must be done before the product is ready for release. In fact, most of the motivation of the next section is to try and estimate just how much more work must be done before the software is ready for release.

Model Fitting The data analysis of the previous section has shown that the entire system, both hardware and software, needs additional testing and debugging. The primary question from the management viewpoint is how long will this take. The marketing position of the company is closely tied to the time of release of the software. Also, the cost of continued field and laboratory testing and debugging can fairly well estimated knowing the additional time required. (The impact on the work force is significant since the company does not maintain a big software shop with many project and programmers who can be rotated from project to project.) We will now use the reliability model described in Eqs. (9.1)-(9.5) to estimate the growth in software MTBF with continued testing.

The first estimate that we can make is to use the historical rule-of-thumb* that the number of errors found during integration testing is about 1% of the number of program object instructions. Thus, E_T is about 0.01 x 50,000 instructions, or 500 errors. In Fig. 10 and Fig. 11 we depict the cumulative number of errors found during field trials and the error discovery rate vs. the number of test hours.

* See Tables 2, 3 on pp. 364, 365 of Ref. 44.

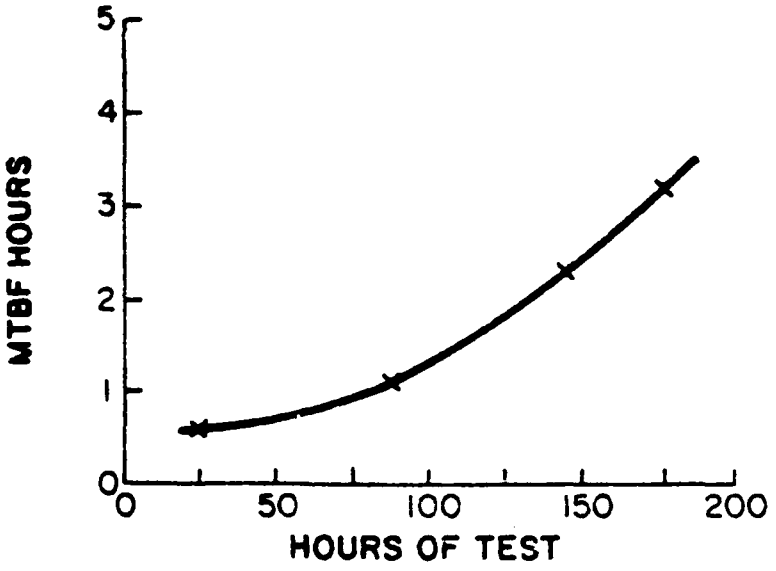


Fig. 9 Increase in Software MTBF During Field Trials.

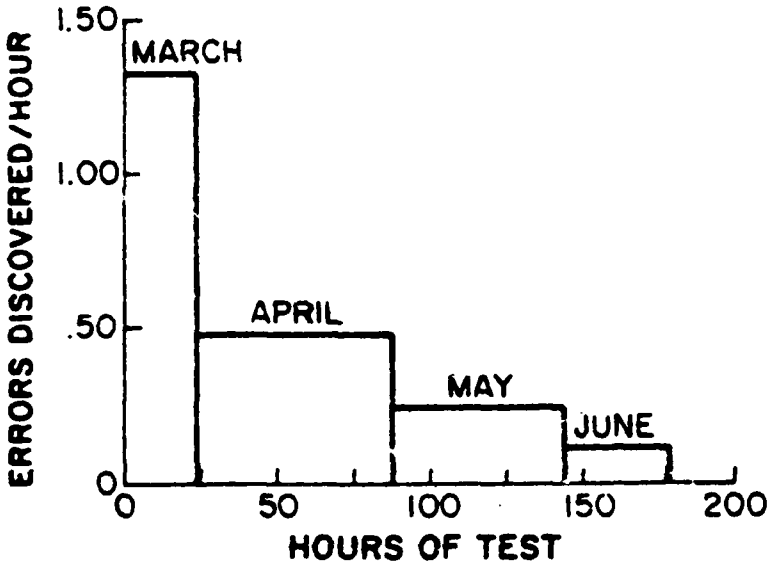


Fig. 10 Decrease in Error Discovery Rate During Field Trials.

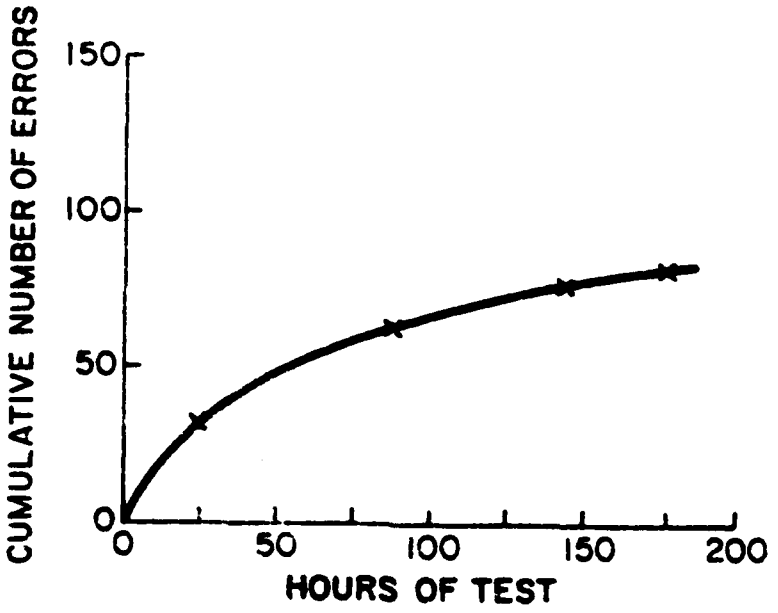


Fig. 11 Cumulative Number of Errors Found During Field Trials.

Clearly, it will take a long time to remove the 500 errors (or even the more optimistic amount of 250 errors) predicted by the rule-of-thumb, if the error discovery rates shown in Fig. 10 continue. Of course the big question is how many additional errors were found and corrected during this period in the development laboratory? Unfortunately, this key set of data was not available, and various assumptions must be made.

Suppose we assume, as a worst case, that no additional errors were found in the development laboratory, that E_T = 500, and that the error discovery rate will be 0.125 errors/hr.* Using these pessimistic values, the additional 421 errors would require 3368 additional hours of testing. Similarly, if we make a best case estimate, we can assume that E_T = 250, that the development laboratory also found 81 errors in addition to those found in field trials, and that the combined error discovery rate of extended field trials and additional laboratory development will be 0.500 errors/hr. Using these optimistic values, the remaining 88 errors will take 176 additional field and laboratory test hours to find. Assuming a crash schedule of ten hours per day and seven days per week, the range of additional time needed is from 2½ weeks to 11 months! Clearly, the range of the above estimates is broad, and we sorely need the missing data on errors found in the development laboratory, and a better estimation procedure. In the next section we make other assumptions about the number of development laboratory errors found, and utilize the model of Eqs. 9.1-9.5, for improved estimation.

To use the reliability model described by Eqs. 9.1-9.5, we must have an accurte record of E_c, and must have a measurement of MTBF at two different points in the development cycle. This leads to two simultaneous equations which are solved for the unknowns, K' and E_T. Following the development in Chap. 5 of Ref. 45, we begin by writing the reciprocal of Eq. 9.5 at two different points.

$$\lambda_1 = K'(E_T - E_c(\tau_1)) \tag{9.6}$$

$$\lambda_2 = K'(E_T - E_c(\tau_2)) \tag{9.7}$$

Solving Eqs. 9.6 and 9.7 we obtain

$$K' = \frac{\lambda_1 - \lambda_2}{E_c(\tau_2) - E_c(\tau_1)} \tag{9.8}$$

$$E_T = \frac{\lambda_1}{K'} + E_{c(\tau 1)} \tag{9.9}$$

* If we assume that the error discovery rate continues to drop, then we probably should assume an even smaller value for error discovery rate for a worst case estimate. This estimate assumes it holds steady.

In order to utilize Eqs. 9.8 and 9.9 to estimate the model parameters, we must know how many errors have been removed from the software at each point in time. Since the data is incomplete in this regard, we have analyzed the data by month for convenience rather than between releases of the software.* In the last three columns of Table 2, we make three different assumptions about error removal:

1. We assume that no errors are found in the development laboratory, and that errors found during the field trials are only fixed at the end of the month. (Of course, if an error shows up later during the field trials, it is listed as an unremoved error, cf. Table 1, until the end of the month following its last occurrence.)

2. We assume that no errors are found in the development laboratory, and that errors found during field trials are fixed during the same month (except for reoccurring ones).

3. Like Assumption 2, however, we assume that the development laboratory finds an equal number of errors distinct from those found in field testing. Thus, Assumption 3 leads to double the number of errors of Assumption 2.

We apply Assumptions 1-3 for the three paired sets of data in Table 2 (March-April, April-May, May-June), use Eqs. 9.8 and 9.9, and arrive at nine sets of values for E_T and K', as shown in Table 3. Note that all the estimates of E_T are significantly smaller than our estimate of 500 errors using the 1% rule-of-thumb.

It is interesting to examine Eq. 9.5 for the case where there is only one error left. At this point the statistical assumptions of the model break down; however, this value, MTBF = $1/K'$, can be viewed as a sort of limiting value. Using the May-June values for Assumption 3, $1/K'$ = 140 hours. If there are a total of 205 errors and 162 have been removed, then 43 remain to be removed. If error removal proceeds at the same rate as error discovery, and error discovery is 0.25/hr., then 172 hours of testing are needed to raise the MTBF to about 140 hours. This is equivalent to 43 four-hour days. The choice of other assumptions would yield a different result.

Another way of estimating the model parameter is obtained by rewriting Eq. 9.6 for any time τ_i and rearranging terms, so that

$$E_c(\tau_i) = E_T - \frac{1}{K'}\lambda_i \qquad (9.10)$$

* Even if we have accurate data, it is often a good procedure to lump the data in an interval for smoothing, rather than treating each error separately, c.f. Ref. 46, pp. 164-170.

Table 3
Determination of Model Parameters Using Eqs. 9.8 and 9.9, and the Data

	Assumption 1		Assumption 2		Assumption 3	
Months	K'	E_T	K'	E_T	K'	E_T
March-April	0.032	53.4	0.0229	99.8	0.0114	200
April-May	0.0143	88.7	0.0357	85.5	0.0179	171
May-June	0.00714	117	0.0143	103	0.00714	205

Table 4
Comparison of the Various Reliability Estimates

Estimate	E_T errors	1/K' hours	Additional time to reach one remaining error days
1% rule-of-thumb and discovery rate = 0.125 errors/hr.	500	-	3368
1/2% rule-of-thumb and discovery rate = 0.500	250	-	178
Table 2, May-June, Assumption 3 and Discovery rate = 0.25	205	140	172
Fig 3c and discovery rate = 0.25	182	76	80

This equation represents a straight line, and the data pairs $E_c(\tau_i)$, λ_i for March, April, May, and June are data points. The parameters are determined from the intercept and slope of a least squares fit to the data. The results appear in Fig. 12a, b, c for Assumptions 1, 2, and 3, respectively. In the case of Assumption 3, the results of the least squares fitting given in Fig. 11c predict values of E_T and $1/K'$ which are somewhat smaller than those given by fitting the May-June data for Assumption 3 in Table 3. The reason is readily apparent from an examination of Fig. 11c. The May-June data fit essentially passes a straight line through the last two data points in the figure. It is clear that such a straight line has a larger slope (larger $1/K'$) and will intersect the y axis at a larger value of E_T.

One of the problems in analyzing this set of data is that there are only four points to work with. If we had more accurate error removal data, we could probably analyze the data in smaller intervals, perhaps once a week. In such a situation, we would have more data points to base our model on. Similarly, additional data on the system at later stages of development would again provide additional data points.

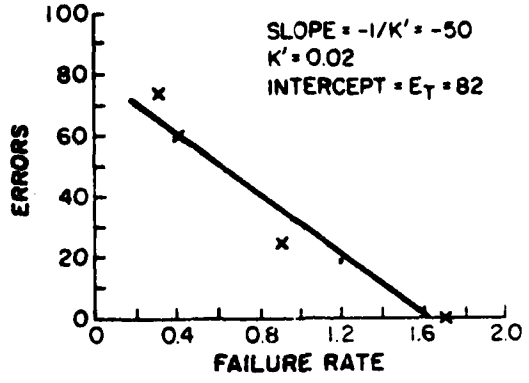
Summary and Conclusions Based on the preceding analysis, we now discuss the six objectives stated in the introduction:

1. Even though the available data was confused and sorely lacking in many respects, by making certain assumptions, we were able to fit a model and make several predictions depending on the assumptions used.

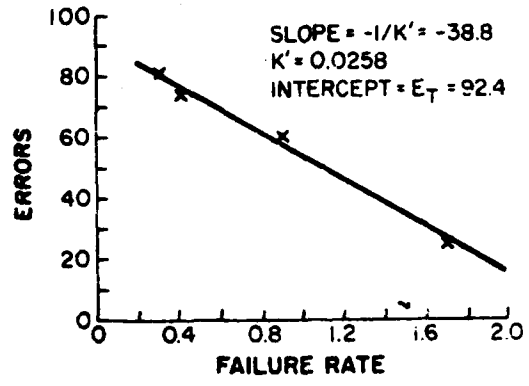
2. Our major assumptions were based on the number of errors detected by the development laboratory, the times at which laboratory and field detected errors were removed, and the anticipated error discovery rate. In each case both an optimistic and pessimistic value were chosen. The results were largely insensitive to the assumptions, since in each case, they predicted substantial and additional testing were needed to have the software ready for release.

3. The various estimates made in this paper can be compared based on the estimated number of total errors, E_T, the "limiting" MTBF = $1/K'$, and the additional test time needed to reach the limiting MTBF. The results are given in Table 4. Note that except for the first worst case estimate, it seems that there should be about 200 total errors, and about 100-200 additional hours of testing will be required to raise the MTBF to about 100 hours.

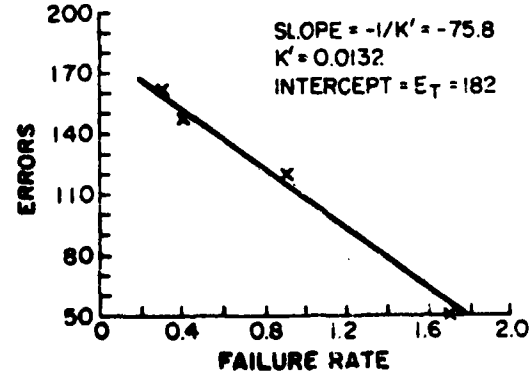
4. Assuming that the results obtained were correct, the range of estimates and their spread are really close enough to make a management decision. The black picture of the worst case estimate can probably be discarded. An important action item



(a) ASSUMPTION 1



(b) ASSUMPTION 2



(c) ASSUMPTION 3

Fig. 12 Least Squares Fit of Table 2 Data to Eq. 9.10 (a) Assumption 1, (b) Assumption 2, (c) Assumption 3. (Using TRS-80 Computer and Radio Shack Statistics Package No. 26-1703.)

is to spend time with the software developers and determine as accurately as possible how many errors were found in the development laboratory, and when and in which release errors found in development or in field trials were removed. Once such data is clarified, the analysis done here can be repeated in a few hours and better estimates can be made. Also of prime importance is a study of the types of testing to be performed during the next few months of field trials. Unless new features and sequences are exercised, the error discovery rate may continue to fall and progress may be slower than predicted. (Also, in light of the decision to replace version A by version B, it is imperative that B be placed under effective configuration control.)

5. It seems that the goal of having a 500-hour MTBF for this software development is unrealistic. In Ref. 43 the author quotes typical MTBF of 3-50 hours. In Ref. 47 Musa measured the MTBF of four different programs, and they ranged from 9-31 hours. It is time that management personnel be made responsible for fixing such goals. These goals should be based on experience with similar systems rather than numbers "picked from a hat" which sound attractive but are unrealistic.

9.7 Application of Shooman's Exponential Model to Field Data.

Preliminary work has begun on fitting field data with Shooman's macro exponential model [43,44]. The objective of the work is to evaluate which of the several parameter estimation techniques devised is best in the sense of ease of computation and accuracy of prediction. A further objective is to generate a modest data base with the values of the parameters E_T and K for all the complete sets of available field data. Recent results, obtained from 4 of the 16 sets provided by J. Musa [48], show a good correlation between actual field experience and the prediction by the Shooman's model.

9.8 Further Results on the Applications of Zipf's Law

9.8.1 Introduction

Some of the most important work on complexity done in the last decade is that generally referred to as software science [3]. Maurice Halstead was the founder of this approach, and most of the work in this area has been carried out by him, his former students, and other investigators. Much work was done on experimental verification of the formulas he developed. Others have attempted to explore more basic probabilistic models and derive similar software complexity metrics from fundamental principles [20].

The first success in relating software science to basic probabilistic models was to show that an operator-operand length formula similar to Halstead's formula, could be obtained by applying Zipf's Law to computer

programs [20]. More recent work has shown that:

(1) A probabilistic model of computer programs involving alternating operators and operands [49] predicts either the Zipf Law length formula or the Halstead length formula, depending on the assumptions made [2].

(2) The Halstead volume measure, V, is actually the information content (Shannon Entropy) of the program [50].

One of the validation studies of software engineering involves the application of the formulas [51] to a set of error data collected by Akiyama [52]. Akiyama's metric, (number of decision + calls), fits the data as well as the Halstead effort measure, [2].

We have found that Halstead's results can be derived with Zipf's Laws as a basis or through the use of a probabilistic model of a program generated by selecting operators and operands according certain probabilistic rules. In addition, we can show that many of the concepts are explained when we compute the Shannon's information measure (i.e., entropy), for the program.

9.8.2 Zipf's Length Formula

The fundamental formula [20], for the operator-operand length (i.e., token length) is given as

$$n = t(0.5772 + \ln t) \tag{9.11}$$

(For definition of n and t, see Table 5). The constant 0.5772 is the Euler's constant.

9.8.3 Estimation of Token Length at the Beginning of Design

The evolving field of software engineering has a great need for theoretical concepts, especially quantitative ones. Thus, a complexity measure such as the operator-operand length given in the preceding sections is of considerable use as a tool for theoretical studies. However, it can play an even more important role if it can be used for estimation of program complexity early in the design process.

One method of initially estimating program length* (number of tokens) is to estimate the number of types. We assume that the analyst initially has a complete description of the problem (requirements) and that a partial analysis and choice of key algorithms has been made (initial specifications and preliminary design). An elementary approach might be to estimate the token size by

(1) Estimate the number of operator types in the programming language which will be used by the programmer.

---

* The reader should remember that we do not include in our token count certain nonexecutable code such as: comments, declares, assembler directives, etc.

(2) Estimate the number of distinct operands by counting input variables, output variables, intermediate variables, and constants which will be needed.

(3) Sum the estimates of step (1) and (2) to obtain the value of t and substitute in Eq. (9-11).

An estimation experiment along these lines was carried out on five different programs [20]. The estimated token lengths (obtained by reading the algorithm and applying the above three steps) differ from the actual lengths (obtained by count from the program) by -3.4%, -15.7%, +26.1%, +2.3%, -32.8% respectively.

9.8.4 Relationship to Software Science

Halstead uses many of the same terms as we did in the preceding sections; however, different notation is used. In order to avoid confusion since we will be using both sets of symbols, a brief glossary of terms to be used in this section is given in Table 5.

Table 5

Comparison of Equivalent Terms

Laemmel-Shooman Development		Halstead Development	
Symbol	Terminology	Symbol	Terminology
t_1	Number of operator types	η_1	Number of unique or distinct operators
t_2	Number of operand types	η_2	Number of unique distinct operands
t	Total operand-operator types $t = t_1 + t_2$	η	Vocabulary $\eta = \eta_1 + \eta_2$
n_1	Number of operator tokens	N_1	Total usage of all the operators
n_2	Number of operand tokens	N_2	Total usage of all the operands
n	Token length $n = n_1 + n_2$	N	Length $N = N_1 + N_2$

9.8.5 Halstead's Length Formula

Halstead's gives a formula for program length based on a combinatorial argument on how operators and operands can be combined to form a program.

$$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \qquad (9.12)$$

where

$N \equiv$ Program length (total operators plus operands)

$\eta_1 \equiv$ Number of operator types

$\eta_2 \equiv$ Number of operand types.

Note that Eq. (9.11) and Eq. (9.12) are of similar form. In fact substitution of $t = \eta_1 + \eta_2$ and $n = N$ into Eq. (1) allows direct comparison.

$$N = (\eta_1 + \eta_2) \times (0.5772 + \ln(\eta_1 + \eta_2)) \qquad (9.13)$$

9.8.6 Comparison of Halstead and Zipf Lengths

We begin our comparison of Eqs. (9.11) and (9.12) by investigating certain limiting cases, and citing the results of a direct numerical comparison for a number of examples. In the case where $\eta_1 >> \eta_2$ Eq. (9.13) reduces to

$$N = \eta_1 (0.5772 + \ln \eta_1)$$

A similar result is obtained if $\eta_2 >> \eta_1$. Furthermore, if the dominating η term is large, the constant 0.5772 can be ignored and the two equations differ by the ratio

$$\frac{\ln \eta_1}{\log_2 \eta_1} = \frac{1}{\log_2 e} = \ln 2 = 0.693$$

Thus, the Zipf length in such a case is about 30% smaller than the Halstead length. For moderate size of the dominating η term, the constant 0.5772 tends to "boost" the Zipf length, reducing the difference.

We now consider the special case where the number of operator types is equal to the number of operand types, that is, $\eta_1 = \eta_2 = \eta$. The ratio of Zipf length to Halstead length then becomes

$$\frac{0.5772 + \ln 2\eta}{\log_2 \eta} = \frac{1.27 + 0.693 \log_2 \eta}{\log_2 \eta}$$

Again, for large η the two measures differ by only ln2 and for moderate valus of η, the compensating constant is 1.27, which should narrow the difference even more than before.

A numerical comparison was made between the length formulas by counting t for 17 examples, substituting in the two equations, and comparing the results with the value of N obtained by counting operators and operands [20]. In some cases the results were optimistic and in some pessimistic and the errors were consistently less then 14%.

9.8.7 An Alternating Operator-Operand Model

In Section 9.8.5 we stated the formula for the Halstead length, Eq. (9.12), without any development. In this section we repeat one of Halstead's original derivations and show that a slight modification leads to the Zipf law length, Eq. (9.11). This development is especially appealing for three reasons:

1. It provides a single derivation which leads to either the Halstead or the Zipf length equation, depending on the assumptions made.

2. It is based on a probabilistic model of how a program can be constructed by alternating operators and operands.

3. It provides another theoretical model which leads to Zipf's laws.

We begin by assuming the following model* for a program:

A program is viewed as a sequence of symbols, made up of alternating operator and operand symbols, chosen from "alphabets" of η_1 and η_2 symbols respectively. The program contains exactly η_1 and η_2 operator and operand symbol types, and we consider the program to be generated by a stochastic process. The character string which represents the program is generated by choosing at random from the alphabet of operators, then choosing at random from the alphabet of operands, and continuing the alternation process. The program generation stops when the last unused operator or operand is chosen for the first time.

9.8.8 Derivation of Zipf's Length Equation From the Model

The length of the character string is calculated by formulating the probability distribution of operator and operand lengths and then computing the expected length (mean length). For simplicity we focus on a single character string composed of an alphabet of η_1 symbols. We observe that as our character string generator proceeds, it generates many sub-

* "Algorithm Dynamics," Maurice Halstead and Rudolf Bayer, Proceedings 1973 Annual ACM Conference, p. 126.

strings made up of k symbols, where $k \leq \eta$. We denote the substring lengths as SL_k. By a substring length we mean the number of new symbols generated before a new alphabet type is encountered. Clearly, the length of the string which uses up all η symbol types is just the sum of the lengths of its constituent substrings.

$$SL_\eta = \sum_{k=1}^{\eta} SL_k \tag{9.14}$$

The expected value operator is written as E( ), and the expected value of a sum of independent random variables is the sum of the expected values.* Thus,

$$E(SL_\eta) = E(\sum_{k=1}^{\eta} SL_k) = \sum_{k=1}^{\eta} E(SL_k) \tag{9.15}$$

The probability that substring k has exactly s symbols is given by the probability that first s-1 symbols are generated from k-1, alphabet types followed by a single letter** generated from the remaining alphabet types. If k=7, then there are 6 alphabet types out of η symbols which have already appeared. Thus for k=7, the probability of a substring sequence of length s is given by

$$P(SL_7) = (\frac{6}{\eta})^{s-1} (1-\frac{6}{\eta}) \tag{9.16}$$

In general, for any value of k, the probability of a substring sequence of length s is given by

$$P(SL_{k+1}) = (\frac{k}{\eta})^{s-1} (1-\frac{k}{\eta}) \tag{9.17}$$

and the expected value of substring length (see [54]), is given by

$$E(SL_{k+1}) = \sum_{s=1}^{\infty} s(\frac{k}{\eta})^{s-1} (1-\frac{k}{\eta}) \tag{9.18}$$

Equation (9.18) can be simplified if we notice that the last term is independent of the summation index, and, the first two terms generate a known

---

* See M. Shooman, "Probabilistic Reliability", p. 404, or most probability texts.

** This means that the last substring is generated when the last unused symbol is picked (i.e. used once). Note the similarity between this assumption and those accompanying Zipf's law.

series

$$\sum_{i=1}^{\infty} i x^{i-1} \frac{1}{(1-x)^2} \tag{9.19}$$

Thus, Eq. (9.18)simplifies to

$$E(SL_{k+1}) = \frac{1}{1-\frac{k}{\eta}} = \frac{\eta}{\eta-k} \tag{9.20}$$

Substitution of Eq. (9.20) into Eq. (9.15) yields an expression for the sequence length.

$$E(SL_{\eta}) = \eta \sum_{k=1}^{\eta} \frac{1}{\eta-k+1} \tag{9.21}$$

A term by term inspection of Eq. (9.21) shows that it is a sum of a descending sequence of terms $1/\eta$, $1/(\eta-1),\ldots,$ $1/1$ which is equivalent to the ascending sequence

$$E(SL_{\eta}) = \eta \sum_{i=1}^{\eta} \frac{1}{i} \tag{9.22}$$

Had Halstead and Bayer desired, at this point in their derivation, they could have approximated the summation and obtained

$$E(SL_{\eta}) \approx \eta(0.5772 + \ln \eta) \tag{9.23}$$

which is a different form of equation (9.11).

9.8.9 Derivation of Halstead's Length Equation From the Model

Halstead and Bayer chose to bound Eq. (9.22) by transforming the summation. Letting

$$i = 2^j$$

then

$$j = \log_2 i$$

and for $i = \eta$, the last equation becomes $j = \log_2 \eta$ and for $i = 1$, $j = \log_2 1 = 0$. Using these substitutions, the summation in Eq. 9.22 becomes

$$E(SL_{\eta}) = \eta \sum_{j=0}^{\log_2 \eta} \frac{1}{2^j} \tag{9.24}$$

The leading terms in the summation above are 1 + 1/2 + 1/4 ... If we recognize the fact that each term is always smaller than unity, then one can bound Eq. (9.24) by assuming that each term $1/2^j \leq 1$, thus

$$E(SL_\eta) \leq \eta \log_2 \eta \qquad (9.25)^*$$

If we return to our previously stated program model, and impose the constraint that operators and operands must alternate, then the expected length is the sum of the expected operator and operand lengths. Thus Eq. (9.25) leads directly to Eq. (9.12), which is the Halstead length equation.

If we remove the restriction that operators and operands must alternate, then $\eta = \eta_1 + \eta_2$, and substitution into Eq. (9.23) yields the Zipf length, [(c.f. Eq. (9.13)]. If we had still retained the restriction that operators and operands alternate, and substituted into Eq. (9.23), we would have obtained another length equation (which must surely give nearly the same numerical answers), namely

$$L = \eta_1 (0.5772 + \ln \eta_1) + \eta_2 (0.5772 + \ln \eta_2) \qquad (9.26)$$

9.8.10 Information Content of a Program

One of the most fundamental results of statistical communication theory is Shannon's information theory. The central formula defines the information content (also known as the entropy H) which has the units of bits. If we have a message which is selected from a set of i messages, each with a probability of occurrence, p_i, then the entropy is given by

$$H = -\sum_{j=1}^{i} p_j \log_2 p_j = \sum_{j=1}^{i} p_j \log_2 (1/p_j) \qquad (9.27)$$

If all the messages are equiprobable, then $p_j = 1/i$, and Eq. (9.27) becomes

$$H = \log_2 i \qquad (9.28)$$

For illustration, suppose that our family of messages is the 16 different binary numbers we can express with a 4-digit binary number. In this case, i=16, and if we assume $p_i = 1/16$, then Eq. 9.28 yields H = 4 bits. This example illustrates the appropriateness of "bits" as the unit for H. As a second example, let us again consider that our messages are the 16 different binary 4-digit numbers; but we now assume that $p_i = 1/2$, $p_2 = p_3 = 1/4$, $p_i = 0$ for i = 4, 5,...,15.

Substitution of these values into Eq. (9.27) yields

---

* Strictly speaking, the upper limit of the summation must still be an integer so it is really the largest integer smaller than $\log_2(\eta+1)$.

$$H = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{1}{4}\log_2 4 + 0 + \ldots = 1.5 \text{ bits}$$

In general, one can show that H is maximized when all the p_i's are equal.

If we drop the restriction that operators and operands must alternate, then we can view a program as a sequence of N symbols (operators or operands). If the probabilities of each symbol are equal, then $p_j = 1/(\eta_1 + \eta_2)$, and the entropy is given by

$$H = N \log_2(\eta_1 + \eta_2) \qquad (9.29)^*$$

We now return to our Zipf's law model of a program and assume that the probabilities p_j are not equiprobable but are given by Zipf's law. It can be shown that [50] the use of the Euller-MacLaurin series allows one to express the summation obtained by substitution in Eq. (9.27) as

$$H = \frac{N}{\ln 2}\left(\frac{(\ln t)^2}{2(\ln t + 7/12)} + \ln(\ln t + 7/12)\right) \qquad (9.30)$$

For say $t > 100$, we can neglect the 7/12 terms, and since $\ln t = \ln 2 \cdot \log_2 t$ Eq. (9.30) can be simplified to give

$$H \sim N \log_2(\sqrt{t} + \ln t) \qquad (9.31)$$

for large t, $\sqrt{t} >> \ln t$, and we obtain

$$H = \frac{N}{2}\log_2 t \qquad (9.32)$$

Note that since $t = \eta_1 + \eta_2$, Eq. (9.32) gives 1/2 the entropy of Eq. (9.29). This is due to the fact that the Zipf law distribution of probabilities has reduced to 1/2 maximum value of H, which the equiprobable distribution yielded.

Halstead proposes that if Eq. (9.29) (or Eq. 9.32) represents the information content (volume) of a program, then the minimum value of H, denoted by H* is obtained if the minimum values of η_1, η_2, and N, η_1^*, η_2^*, and N* are used. He defines the minimum number of operators to be two, a function which does all the work of the program, f(), and an assignment symbol (equal sign). Alternately we could consider the two operators as PRINT (f()). The minimum number of operands are the sum of the required input and output variables. Since we only use each operator and operand once $N_1 = \eta_1$ and $N_2 = \eta_2$, thus

$$\text{Min } \eta_1 = \eta_1^* = 2 \qquad (9.33)$$

$$\text{Min } \eta_2 = \eta_2^* = \text{sum of input and output variables} \qquad (9.34)$$

$$\text{Min } N = \eta_1 + \eta_2 \qquad (9.35)$$

---

* In his work on "Software Physics", Halstead defines a quantity called volume, denoted by V, which has the same formula

Substitution of Eqs. (9.33)-(9.35) into Eqs. 9.29 and 9.32 yields

$$V^* = (2 + \eta_2^*) \log_2(2 + \eta_2^*) \tag{9.36}$$

$$H^* = V^*/2 \tag{9.37}$$

A direct extension of the concept of minimum information content or volume is to define the level, ℓ, of a program as the ratio of minimum volume to actual volume

$$\ell = \frac{V^*}{V} = \frac{H^*}{H} \tag{9.38}$$

Halstead proposes an effort measure, E, which is the number of mental discriminations needed to develop a program. He postulates that this measure should be proportional to the program volume and inversely proportional to the level, thus

$$E = \frac{V}{\ell} \tag{9.39}$$

Using Eq. 9.38 and the fact that H = V/2, (for a Zipf law distribution), we obtain alternate expressions for E as

$$E = \frac{V^2}{V^*} = \frac{2H^2}{H^*} \tag{9.40}$$

9.8.11 Correlation Between the Proposed Metrics and Experience.

In this section we investigate the correlation between the number of program errors or man-months of development time and:

(1) the token length;
(2) the volume/information;
(3) the effort metric E.

The complexity measures developed in the preceding sections are useful in two ways: (1) they provide metrics which allow comparison of the relative complexity of two different designs or algorithms, (2) when multiplied by an appropriate proportionality constant, they should provide estimates and predictions for the number of errors and manpower required for software development.

Number of Errors vs. Complexity We begin by discussing the relationship between the number of errors and our measures of complexity. Unfortunately, there are only a few sets of complete error data in the literature. We use the set of data collected by Akiyama [51] for the correlation studies which follow. Akiyama's machine language program was about 25,000 assembly language instructions long and contained the 9 modules shown in Table 6. The number of bugs (errors) found in each module during development is given in the third column of the table.

The data collected by Akiyama is sufficiently complete to warrant detailed exploration, and four different hypotheses will be tested:

Table 6 Raw Data from Akiyama ([51])

Module	Machine lang. Statements	Bugs	Decisions	Calls	Desisions+ Calls
MA	4,032	102	372	283	655
MB	1,329	18	215	44	249
MC	5,453	93 (146)*	552	362	914
MD	1,674	26	111	130	241
ME	2,051	71	315	197	512
MF	2,513	37	217	186	403
MG	699	16	104	32	136
MH	3,792	50	233	110	343
MX	3,412	80	416	230	646

*Akiyama gave two different values for this module

Table 7 Information Derived from Akiyama's Data

Module	N = Twice the number of machine language statements	η_1	η_2	H	E
MA	8,064	471	442	79.3×10^3	170.3×10^6
MB	2,658	180	176	22.5	15.3
MC	10,906	610	574	111.3	322.6
MD	3,348	231	201	29.3	28.2
ME	4,102	366	138	36.8	100.2
MF	5,026	322	287	46.5	65.5
MG	1,398	131	76	10.8	6.5
MH	7,584	252	603	73.9	58.5
MX	6,824	433	357	65.7	135.9

1. Length Hypothesis - The number of bugs is proportional to the number of machine language statements. Since each machine language statement contains one operator and one operand, the number of machine statements is just 1/2 the number of tokens (operators + operands). Thus a study of bugs vs. machine language statements or token length are equivalent since two lengths are directly proportional to another.

2. Information Hypothesis - The number of bugs is proportional to the information content of the program, where information content (Halstead volume), is calculated from Eqs. (9.29) or (9.32). (The two equations are identical, except for the factor of 1/2, and that will be absorbed in the experimentally determined constant of proportionality).

3. Halstead Effort Measure - The number of bugs is proportional to effort in programming, E. This hypothesis was explored by Funami and Halstead.[52]

4. Akiyama's Hypothesis - The number of bugs is proportional to the number of decisions plus subroutine cells.

9.8.12 Use of Akiyama's Data for Correlating Errors and Complexity Measures

Akiyama's raw data, given in Table 7, provides us with the appropriate quantities to test the Length Hypothesis and Akiyama's Hypothesis. In order to calculate H and E so as to test the other two hypotheses, we must make some assumptions. First we assume that each machine langauge statement contains one operator and one operand, thus N = twice the number of statements. (See second column of Table 7). To compute the number of unique operators, η_1, Halstead assumed that it was equal to the sum of the number of machine language instruction types, the number or program decisions, and the number of unique program calls. He guessed that there were 64 types of machine language instructions, and that only 1/3 of the subroutine calls were unique and arrived at the formula

$$\eta_1 = \text{decisions} + (\text{calls}/3) + 64 \qquad (9.41)$$

The third column of Table 7 is calculatd by use of the above formula. Knowing η_1 and N, we are able to calculate η_2 by substitution in Eq. (9.12). This provides the data necessary for computation of H (see Table 7). Finally, if we assume that $\eta_2^* = \eta_2$, we can compute E. The four hypotheses are tested in Figs. 13-16. Clearly the first two hypotheses fit the data fairly well, but the third and fourth hypotheses yield better fits. In fact, Halstead's effort measure yields an excellent fit except for the MC module. (Note, since Akiyama speaks of two values for the MC we plot one as MC and the other as MC').

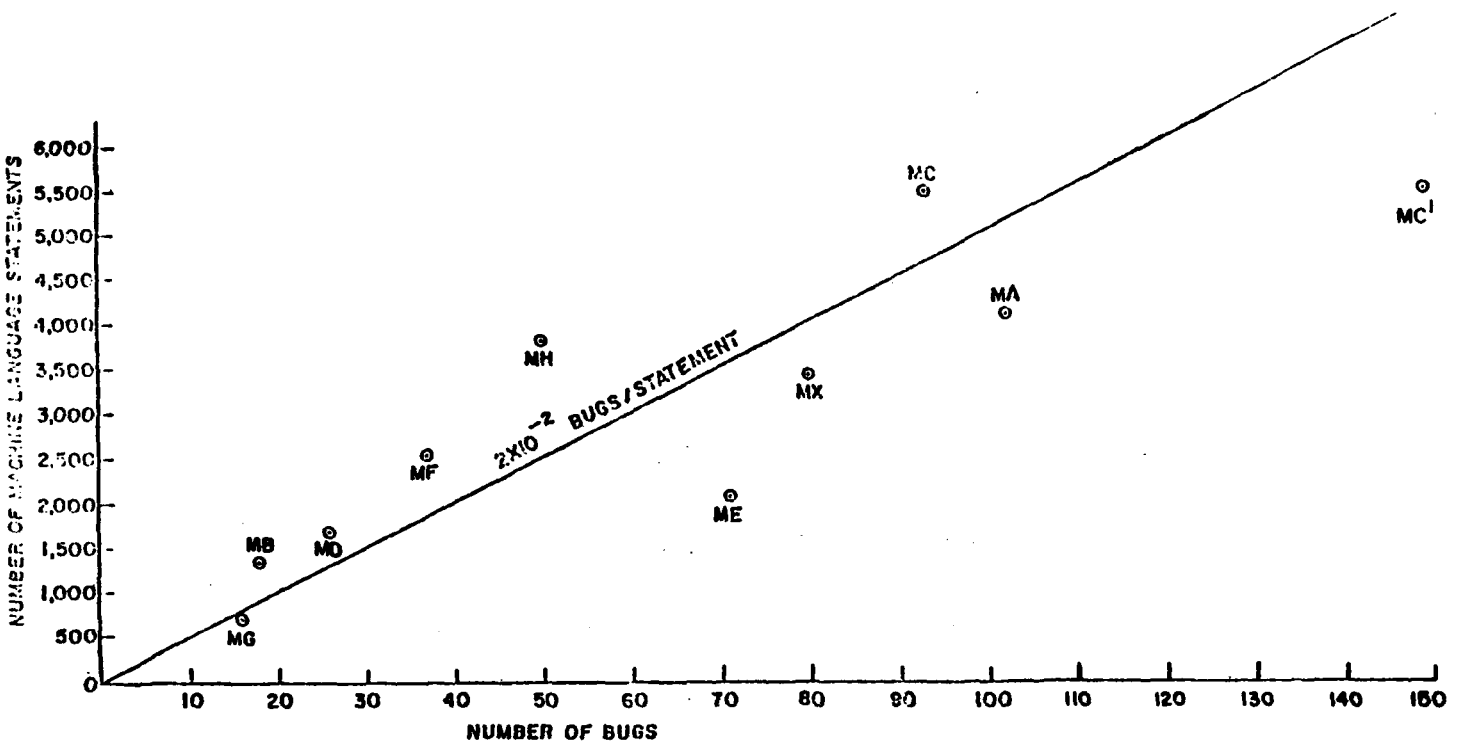


Fig. 13 Number of Bugs vs. Number of Machine Language Statements

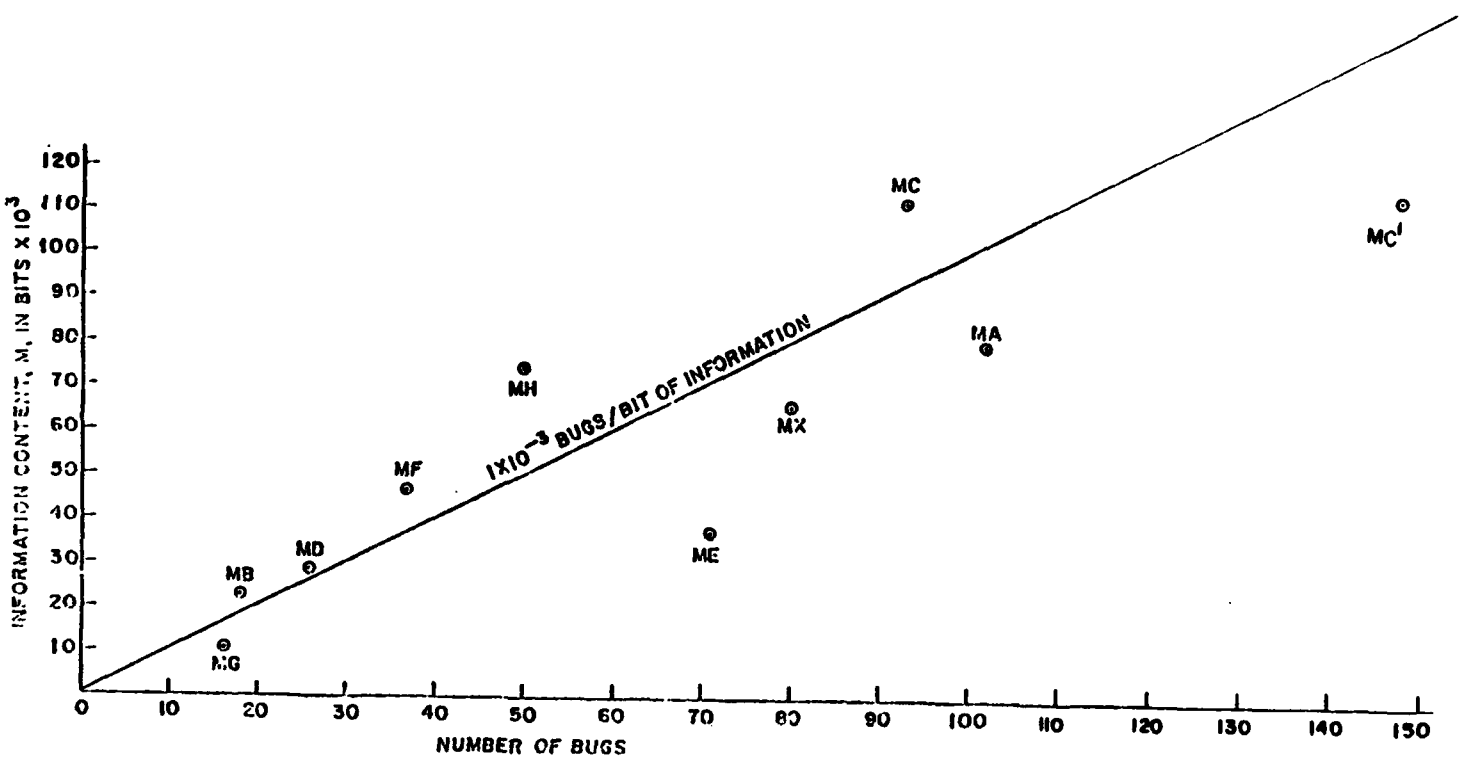


Fig. 14 Number of Bugs vs. Information Content in Bits

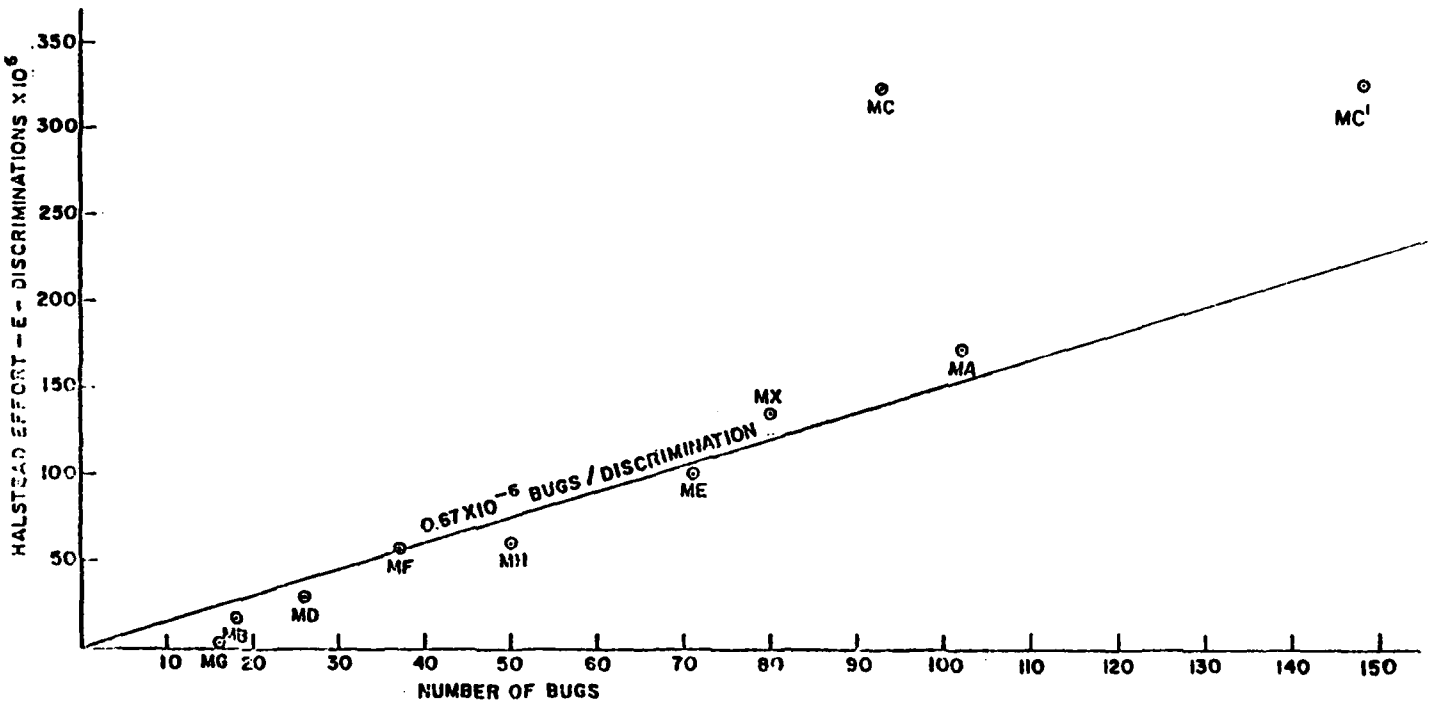


Fig. 15 Number of Bugs vs. Halstead Effort - E

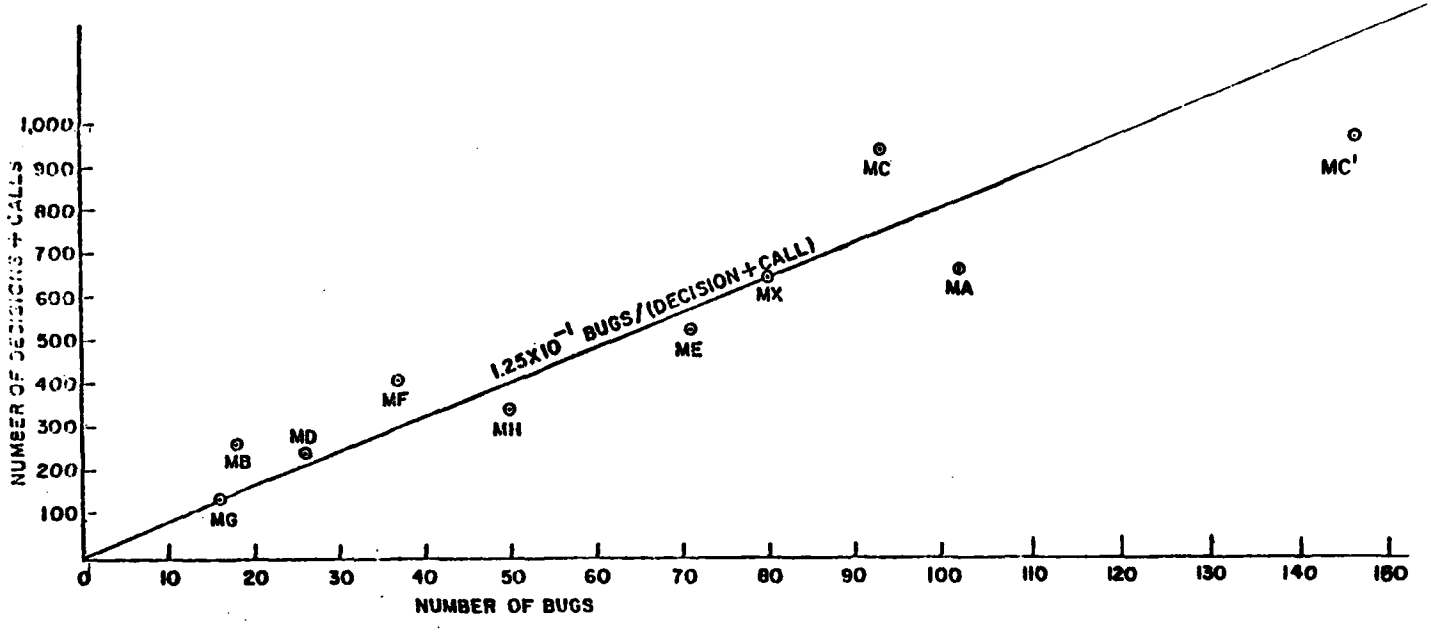


Fig. 16 Number of Bugs vs. Akiyama's Measure

9.8.13 Correlation of Hypotheses with data

A best straight line, ($y = mx + b$), was fitted to Figs. 13 - 16. Both the values for MC, 93 and 146, were used. The resulting values of m, b, and the correlation coefficient ρ are given in Table 8. The straight lines drawn through the data in Figs. 13 - 16, are not the least squares lines, but were chosen by eye to: (1) pass through the origin; (2) have an even reciprocal slope; (3) fit the data points reasonably well. A set of proportionality constants for the lines given in the figures is calculated in Table 9.

9.8.14 Summary and Conclusions

The major results obtained are:

1. It was shown both analytically and experimentally that the Zipf law length formula and the Halstead length formula yield answers which agree within 15% for the examples tested.

2. A probabilistic model of a program first suggested by Halstead can be used to derive the Halstead formula by placing an upper bound on the sum of a series or the Zipf formula by approximating the sum. From a theoretical viewpoint the latter seems better.

3. By introducing the concept of program information content (entropy), we are able to show that the quantity defined as volume by Halstead is the information content of a program if we assume an equiprobable distribution of operators and operands. Assuming a Zipf law distribution reduces the information content by a factor of 2.

4. Correlation of several measures with Akiyama's data on program errors show good agreement for all measures, but the best correlation is for the effort function and for Akiyama's measure.

More work needs to be done to compare the above results with those obtained by studying other data bases.

9.9 Cost Estimation

The work on cost estimation followed several avenues. Initially, the seven prominent methods (Price S, Doty, Walston-Felix, Wolverton, Aron, SDC, and Nelson) were compared by predicting the development costs with each method for the same program. Secondly, an inflation correction was incorporated and a corrected prediction was obtained. The cost in thousands of dollars varied from \$625K to \$1,525K the mean value was \$1,096K and be variance \$303K. Other features of this work included:

Table 8

Least Squares Fit of a Straight Line for the Four Hypotheses

	Hypothesis	Correlation Coefficient ρ	Slope m	Intercept b
1.	Bugs Proportional to Machine Language Instructions			
	MC = 93	0.832	38.5	666
	MC' = 146	0.896	31.2	879
2.	Bugs Proportional to Information Content			
	MC = 93	0.828	811	8,487
	MC'= 146	0.900	865	12,572
3.	Bugs Proportional to Effort E			
	MC = 93	0.853	2,595,000 - 41,835,000	
	MC'= 146	0.982	2,251,000 - 36,251,000	
4.	Bugs Proportional to Calls plus Decisions			
	MC = 93	0.923	7	72
	MC'= 146	0.976	5.58	116.9

Table 9

Comparison of Proportionality Constants

1 statement = 2 tokens

2×10^{-2} bugs/statement ÷ 2 tokens/statement = 1×10^{-2} bugs/token

1×10^{-2} bugs/token ÷ 1×10^{3} bugs/bit = 10 bits/token

1×10^{-2} bugs/token ÷ 0.67×10^{-6} bugs/discrimination = 1.5×10^{4} disc./token

1×10^{-2} bugs/token ÷ 1.25×10^{-1} bugs/(decision + call) = 8×10^{-2} (dec. + call)/token

a. Fitting of the Putnam model to safeguard manpower data.

b. Evaluation of life cycle cost fidelity of the Putnam model with known total costs and build up rate estimated from first year data only.

c. A statistical justification for the superiority of bottom-up estimation over top-down estimation. (i.e., bottom-up estimates are closer to actual costs).

Further discussion, details, and conclusion appear in Reference [55].

9.10 Programming Methods for Low-Error Content

The main body of this work is described in volume 3 of this report. Further work focussed on automatic proramming techniques, began several years ago (and reported in progress reports proceeding the current contract), and continued on a small-effort basis.

The automatic programming contains currently a relatively large pool of programs. Further experiments were performed using this pool of programs. The result will appear in a future comprehensive report.

9.11 Testing

Our previous work on testing was based on graph and flow chart models of programs [26]. The automatic test driver traversed all graph paths in one program and is described in [34]. The statistical approach to testing comprises volume 4 of this report. Several other efforts have been initialed and are listed below:

a. The previously developed techniques for counting and identifying graph paths for loopless programs have been generalized to included programs with loops.

b. The work on automatic test drivers is being broadened to include other languages (e.g. PASCAL and Ada), and additional features.

c. Preliminary analysis has begun on the design of a natural test driver. Such a test driver will transverse all (or a subset of) the actual program paths in the flow chart. This improvement reduces the number of paths to be listed by eliminating graph paths which cannot be reached, thus decreasing test effort and increasing realism.

10. REFERENCES

1. M. L. Shooman, "Software Reliability: Measurement and Models," 1975 Annual Reliability and Maintainability Symposium.

2. M. L. Shooman, "Software Engineering: Reliability, Design, Management," Notes for CS 606, Polytechnic Institute of New York, Dept of Elec. Engineering, Fall 1977, McGraw-Hill Book Co., New York 1981.

3. M. H. Halstead, "Software Science," Elsevier, North-Holland, New York, New York, 1977.

4. F. Akiyama, "An Example of Software System Debugging," Information Processing 71, North-Holland Publishing Co., New York 1972.

5. M. L. Shooman and H. Ruston, "Summary of Technical Progress, Investigation of Software Models," RADC-TR-79-188, July 1979.AD#A073639.

6. M. L. Shooman, "Probabilistic Models for Software Reliability Prediction," Statistical Methods for the Evaluation of Computer System Performance, Frieberger, Editor, Academic Press, New York, 1972.

7. J. Jelinski and P.B. Moranda, "Software Reliability Research," same source as 6.

8. J. Musa, "A Theory of Software Reliability and It's Application, IEEE Trans. on Software Engineering, Vol. SE-2, No. 3, Sept. 1975, p. 312.

9. B. Littlewood, "A Bayesian Reliability Growth Model For Computer Software," Record 1973 IEEE Symposium on Computer Software Reliability, pp. 70-77.

10. M. L. Shooman and S. Natarajan, "Effect of Manpower Deployment and Error Generation on Software Reliability," Proceedings MRI Symposium on Computer Software Engineering, New York City, April 1976.

11. D. K. Lloyd and M. Lipow, "Reliability Management Methods. and Mathematics," 2nd Edition Published by the authors, 201 Calle Miramar, Redondo Beach, CA 90277.

12. M. L. Shooman, "Structural Models for Software Reliability Prediction," Second National Conference on Software Reliability, October 1976, San Francisco.

13. A. N. Sukert, "An Investigation of Software Reliability Models," Proceedings 1977 Annual Reliability and Maintainability Symposium, IEEE, New York, N. Y. p. 478ff..

14. ITT Research Institute: "Software Engineering Research Reviews - Quantitative Software Models," Data and Analysis Center for Software, Rome Air Development Center, Griffiss AFB, N. Y., March 1979.

REFERENCES (Cont'd)

15. G. J. Schick and R. W. Wolverton, "Analysis of Competing Software Reliability Models," IEEE Transactions on Software Engineering Vol. SE-4.

16. A. K. Trivedi and M. L. Shooman, "A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters." Proceedings 1975 International Conference on Reliable Software, April 1975.

17. L. A. Belady, "On Software Complexity," Proceedings of the Workshop on Quantitative Software Models, Kiamesha Lake, N. Y., Oct. 9-11, 1979, IEEE New York.

18. B. Curtis, et. al., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," IEEE Transactions on Software Engineering, 1979, pp. 96-104.

19. B. Curtis, "In Search of Software Complexity," same source as 17, pp. 95-106.

20. A. Laemmel and M. L. Shooman, "Statistical (Natural) Language Theory and Computer Program Complexity," POLY EE/EP-76-020, SMART 107, August 1977.

21. Henry Ruston, "The Polynomial measure of Complexity," POLY EE 79-057, SRS 117, Volume 2, Polytechnic Institute of N. Y., Sept. 1979.

22. M. H. Halstead, "Software Physics: Basic Principles, IBM Research Report RJ 1582, T. J. Watson Research Center, Yorktown Heights, N. Y., 1975.

23. M. Lipow, "Application of Algebraic Methods to Computer Program Analysis," Report TRW-55-73-10, TRW Redondo Beach CA., May 1973.

24. S. Mohanty, "Models and Measurements for Quality Assessment of S oftware," Computing Surveys, Vol. 11, No. 3, September 1979.

25. E. F. Miller, "Automatic Generation of Test Case Data Sets," Proceedings of the IEEE Computer Software and Applications Conference, Chicago, No. 8-11, 1977.

26. G.S. Popkin and M. L. Shooman, "On the Number of Tests Necessary to Verify a Computer Program," POLY EE 78-047, SRS 113, Polytechnic Institute of N. Y., June 1978.

27. A. Laemmel, "A Statistical Theory of Computer Program Testing," POLY EE 80-004, SRS 119 Volume 4, Polytechnic Institute of N. Y., June 1980.

REFERENCES (Cont'd)

28. F. Brooks, "The Mythical Man-Month, Addison-Wesley Pub. Co., Reading, Mass. 1975.

29. R. C. Tausworthe, "Standarized Development of Computer Software," Prentice-Hall, New Jersey, 1977.

30. M. L. Shooman and A. Kershenbaum, "Models for the Management of Software," in Summary of Technical Progress, RADC-TR-79-188, pp. 33-35, July 1979.

31. A. G. Cormier, "A Quantitative Analysis of the Effect of Organizational Structure on Software Engineering Management," Master of Science Thesis, Polytechnic Institute of New York, 1980.

32. M. L. Shooman, "Software Reliability Data Analysis and Model Fitting," Proceedings of the Workshop on Quantitiative Software Models, Kiamesha Lake, N.Y., pp. 182-189, IEEE, Oct. 1979.

33. Workshop on Quantitative Software Models for Reliability, Complexity and Cost, IEEE, October 1979.

34. D. L. Baggi and M. L. Shooman, "Software Test Models and Implementation of Associated Test Drivers," RADC-TR-80-45, March 1980.AD#A065004.

35. M. Klerer, "Experimental Study of Two-dimensional Language vs FORTRAN for First-Course Programmers," SRS - 118, Volume 3, Polytechnic Institute of N. Y., Jan. 1980.

36. A. Laemmel, "Study of Recursive Function Theory and Its Application to Program Complexity," SMART 108-C POLY EE/EP 77-037 Polytechnic Institute of N. Y., May 1978.

37. T. McCabe, "A Complexity Measure," IEEE Trans. on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp. 308-320.

38. M. L. Shooman and H. Ruston "Final Report-Software Modeling Studies," Polytechnic Institute of New York, Sept. 1977, pp. 27-29.

39. A. Kershenbaum, "Software Management Models: A Graph Theoretic Approach to System Morphology," Unpublished Summary, presented at the SOFTY Research Meeting, Feb. 5, 1979, Polytechnic Institute of New York, Farmingdale, N. Y.

40. J. B. Synnott, III, (Bell Laboratories), "Managing Software Development - Requirements to Delivery," Proceedings, Computer Software and Application Conference, 78, Palmer House, Chicago, p. 19.

41. E.C. Schleh, "Managing for Success: Capitalizing on Each Individual" IEEE Engineering Management Review, Volume 7, Number 4, December 1979, pp. 33-41.

REFERENCES (Cont'd)

42. S. J. Amster, et al, "An Experiment in Automatic Quality Evaluation of Software," Proceedings of the Polytechnic Symposium on Computer Software Engineering, Polytechnic Ress, Brooklyn, N. Y. 1976, pp. 171-197.

43. M.L. Shooman, "Probabilistic Models for Software Reliability Prediction," published in "Statistical Computer Performance Evaluation," Walter Freiberger Editor, Academic Press, New York, 1972, pp. 485-497.

44. _____, "Software Reliability," published in "Computing Systems Reliability," T. Anderson and B. Randell Editors, Cambridge University Press, New York, 1979.

45. _____, "Software Engineering: Design, Reliability, Management," McGraw-Hill Book Co., New York, 1980.

46. _____, "Probabilistic Reliability: An Engineering Approach," McGraw-Hill Book Co., New York, 1968.

47. J.D. Musa, "A Theory of Software Reliability and Its Application," IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, September 1975, p. 313.

48. _____, Private Communication to M. L. Shooman, listing excertion true data for 16 projcts, Jan. 1980.

49. M. Halstead and R. Bayer, "Algorithm Dynamics," Proceeding 1973 Annual ACM Conference, p. 126.

50. A. E. Laemmel, Unpublished Memorandum, Dec. 1976.

51. F. Akiyama, "An Example of Software System Debugging," Proceedings of the IFIP Congress, 353-58 (1971).

52. Y. Funami and M. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data, Proceedings of the Symposium on Computer Software Engineering, Polytechnic Press, Brooklyn, April 20-22, 1976.

53. G. K. Zipf, "The Psycho-biology of Language: An Introduction to Dynamic Philogy," First Edition 1935 by Houghton Miffin Co. Boston, Paper Back Edition, MIT Press, Cambridge Mass., 1965.

54. M.L. Shooman, "Probabilistic Reliability: An Engineering Approach," McGraw-Hill Book Co., New York, 1968, p. 55.

55. _____, "Software Cost Models," Proceedings of the Workshop on Quantative Software Models, Kiamesha Lake, N.Y., pp. 1-19, IEEE, Oct. 1979.

11.0 Professional Activites

The results of the research described in the preceding safes, have been disseminated in preliminary and completed forms both orally and in writing. The professional activities are grouped below under the following categories: (1) Papers, (2) Reports, (3) Symposia and Workshops, (4) Talks and Seminars, (5) Books, (6) Technical Committees, and (8) Professional Awards.

11.1 Papers

In the following we list the papers published in Journals and Conference Proceedings during the period of this contract.

1. D.L. Baggi and M.L. Shooman, "An Automatic Driver for Pseudo Exhaustive Software Testing," Proc. 1978 Spring Computer Conference, Feb. 1978.

2. M.L. Shooman, "Safety Metrics and Human Operator Control of Complex Systems," Proc. 4th Int. System Safety Conference, pp. 161-166, San Francisco, July 1979.

3. M.L. Shooman, "Software Reliability," in Computing Systems Reliability," T. Anderson and B. Randell editors, Cambridge University Press, New York, 1979.

4. M.L. Shooman, "Software Cost Models," Proceedings of the Workshop on Quantitative Software Models, IEEE, pp. 1-19, Oct. 1979.

5. M.L. Shooman, Software Reliability Data Analysis and Model Fitting," Proceedings of the Workshop on Quantitative Software Models, IEEE, pp. 182-189, Oct. 1979.

6. D.L. Baggi, "Models of Automatic Drivers for Pseudo-Exhaustive Software Testing," Proceedings of the Workshop on Quantitative Software Models, IEEE, pp. 214-223, Oct. 1979.

7. H. Ruston, "The Polynomial Measure of Complexity," Submitted to IEEE Transactions on Software Engineering, October 1979.

8. E. Berlinger, "An Information Theory Based Complexity Measure," 1980 National Computer Conference.

9. S. Kao and M.L. Shooman "Probabilistic Approaches to the Combination of Loads in Structural Design," Proceedings of the Annual Reliability and Maintainability Symposium, Jan. 1981

10. M.L. Shooman and R. Schmidt, "Fitting of Software Error and Reliability Models to Field Failure Data" Invited paper, Proceeding of the Conference or Applied Probability - Computer Science TIMS/ORSA Fall 1981.

Papers (Cont'd)

11. R.F. Juels, "Fault Mending Considerations of Reversible Directed Links," Submitted to the 1981 International Symposium on Fault-Tolerant Computing.

11.2 Reports

The following reports were published during the duration of this contract

1. M.L. Shooman and S. Sinkar, "Generation of Reliability and Safety Data by Analysis of Expert Opinim," Polytechnic Institute of N.Y., Jan. 1978.

2. A. Laemmel, "Study of Recursive Function Theory and its Applications to Program Complexity," SMART 108-C, Poly EE/EP-77-037 May 1978

3. G.S. Popkin and M.L. Shooman, "On the Number of Tests Necessary to Verify a Computer Program," RADC-TR-78-229, November 1978. AD#A089997.

4. M.L. Shooman and H. Ruston, "Software Modeling Studies," Progress Report 43, Report R-452.43.785 pp. 423-448 Polytechnic Institute of N.Y., Nov. 1970.

5. M.L. Shooman, "Mathematical Models of Human and Ship Responses for Minimum Time Turns," CADRF.40-7901-01 National Maritime Research Center, Kings Point, N.Y. Feb. 1979.

6. M.L. Shooman and H. Ruston, "Summary of Technical Progress, Investigation of Software Models," RADC-TR-79-188, July 1979.

7. H. Ruston, "The Polynomial Measure of Complexity," POLY EE 79-057 SRS 117, Sept. 1979.

8. M.L. Shooman and H. Ruston, "Software Modeling Studies," Progress Report 44, Report-452.44-79 Polytechnic Institute of N.Y., Nov. 1979.

9. M. Klerer, "Experimental Study of a Two-Dimensional Language vs Fortran for First-Course Programmers, SRS-118, POLY EE-80-001, Jan. 1980.

10. M.L. Shooman and H. Ruston Final Report, Software Modeling Studies SRS 120, POLY EE 80-006 Jan. 1980.

11. M.L. Shooman, "Models at Helmsman and Pilot Behavior for Maneuvering Ships", Grumman Data Systems Corp. Report, Jan. 25, 1980.

12. C. Marshall, "Incentivizing Availability Warranties," POLY-EE Report No. 80-002, EE-126 March 1980.

Reports (Cont'd)

13. D.L. Baggi and M.L. Shooman "Software Test Drivers," RADC-TR-80-45, March 1980.

14. A.E. Laemmel, "A Statistical Theory of Computer Program Testing," SRS-119, POLY EE 80-004, June 1980.

11.3 Symposia and Workshops

Because of the crucial role of quantative models in the software field, and the high relevance to the objectives of this contract, we organized a workshop on such models. This workshop took place on October 11 and 12 at the Concord Hotel, Kiamesha Lake, N.Y.

The workshop served as the focal point for the nearly 100 leading workers in the field, who came from England, France, Canada, and all parts of the United States. The technical sessions, concentrating on costing, complexity, and reliability began with a brief tutorial by an expert in field. The program of each session contained papers reporting on the advances in the field. The final session critically assessed the state of the art and the needs of each area during the next decade.

The resulting proceeding were published by IEEE and are available from the Computer Society.

11.4 Talks and Seminars

M.L. Shooman

1. "Software Reliability Models", Computer Science Seminor, Queens College, May 18, 1978.

2. "Software Engineering Models," ACM Spring Lecture Series, New York Chapter, May 18, 25 1978.

3. "Software Models-Some Applications," Syracuse University/RADC Workshop, Minnowbrook, Sept. 1978.

4. Discussant, Software Reliability Research, Graduate Reading Course, University of Maryland, Sept. 23, 1978.

5. "Software Engineering Models", Computer Science Seminar, University of Maryland, Oct. 1978.

6. Software Testing and Development Industry Seminars, CDC Corporation, Denver, Colorado, Washington, D.C.,Pittsburg, Pa., 1978-1979.

7. "Availability and Reliability Modeling," Industry Seminars Stata, Metrics, Princeton, N.J. 1978, Elizabeth, N.Y. 1979, New Brunswick (Rutgers Univ.), 1979.

Talks and Seminars - Cont'd (M.L. Shooman)

8. Invited Lecturer For 7 Sessions, Course on Software Engineering, IBM System Research Institute, New York, 1979.

9. "Software Development Techniques," Invited Seminar, Martin-Marieta Internal Workshop, Feb. 1979.

10. "Safety Metrics and Human Operator Control of Complex Systems," Proc. 4th International System Safety Conference, San Francisco, July 1979.

11. "Software Reliability and Software Complexity", Invited Lecturer, Course on Advanced Computer Systems Reliability, University of California Santa Cruz, July 1979.

12. Introductory Lecturer and Organizer of 5 Seminars in Reliability for Department of ME/AERO, Polytechnic, Fall 1979.

13. "Software Reliability Models", Invited Tutorial Lecturer, Annual Reliability and Maintainability Symposium, Jan. 1979, 1980, 1981.

14. "Research Progress, Software Modeling Studies", Rome Air Development Center, Dec. 1978, Dec. 1979, Aug. 1980.

15. "Trends in Reliability and Quality Control Education", ASQC Seminar, Dec. 1979.

16. "Software Engineering-State of the Art," Industry Seminars, Hazeltine Corp., Greenlawn, N.Y. Spring 1980.

17. "Modern Methods in Electrical Engineering and Computer Science," Industry Seminars, PRD Corp. Syosett, N.Y. Fall 1980.

18. "The Information Theoretic Basis of Software Design", Syracuse University/RADC Workshop, Minnowbrook, Aug. 19-22, 1980.

19. "Software Engineering Models-An Assessment", Electrical Engineering and Computer Science Seminar, Renssaleaer Polytechnic Institute, Nov. 1980.

H. Ruston

1. "Software Models-Some Applications," Syracuse University/RADC Workshop, Minnowbrook, Sept. 1978.

2. "Research Progress - Software Modeling Studies," Rome Air Development Center Dec. 1978, Dec. 1979, Aug. 1980.

3. "Structured Design," Polytechnic Industry Seminars, May 1979, N.Y. City.

Talks and Seminars (Cont'd) - H. Ruston

4. "Pascal Programming," Polytechnic Industry Seminars May 1979, N.Y. City, June 1979, Bethesda, Md., August 1979, Yonkers, N.Y.

5. "Pascal - An Introduction," Industry Seminar, Loral Electronics Systems, Yonkers, N.Y., Aug. 1979.

6. "Software Costs-Reduction and Control," William Patterson College of N.J., March 1980.

7. "Software Engineering-State of the Art," 6 Industry Seminars, Hazeltine Corp., Greenlawn, N.Y. Spring 1980.

8. "The Polynomial Measure of Complexity," Seminar, Polytechnic Institute of N.Y., May 1980.

9. "Geometrical Complexity - A Comparison" Syracuse University/ RADC Workshop, Minnowbrook, Aug. 19-22, 1980.

10. "Modern Methods in Electrical Engineering and Computer Science," 16 Industry Seminars, PRD Syosett, N.Y., Fall 1980.

11.5 Books

M.L. Shooman:

1. Chapter on "Software Engineering", In Book "Computing System Reliability," Editor Brian Randell, Cambridge University Press, 1979.

2. "Software Engineering: Design, Reliability, Management," McGraw-Hill Book Co., 1981.

3. "Electrical Engineering Handbook," J. Wiley and Sons, N.Y. 1981.

H. Ruston

1. "Programming with PL/I," McGraw-Hill Book Co., 1978.

2. "Electrical Engineering Handbook," J. Wiley and Sons, N.Y. 1981.

11.6 Technical Committees

M.L. Shooman

1. Member of Advisory Committee, Reliability Society, 1972-1978.

2. Member of Advisory Committee, Software Engineering Committee, Computer Society, 1975-Present.

Technical Committees (Cont'd) - M.L. Shooman

3. Technical Program Chairman and Session Chairman, IEEE/POLY Workshop on Quantitative Software Models, Kiamesha Lake, N.Y. Oct. 1979.

4. Invited Attendee, IEEE/NRC Conference on Nuclear Safety, Myrtle Beach, S.C., Dec. 1979.

5. Member and Acting Administrator, IEEE Standard- 500, The Collection, Analysis, and Publication of "Electrical, Electronic, and Sensing Component Reliability Data for Nuclear-Power Generating Stations", 1976-1978.

6. Member of IEEE Computer Society Software Engineering Standards Committee, 1976-1979.

7. Member of Long Island Section Fellows and Awards Committee. 1979-Present.

8. Member of IEEE Edison Medal Selection Committee, 1980.

H. Ruston

1. Member of IEEE Committee on PASCAL Standards - 1979-Present.

2. Chairman, IEEE/POLY Workshop on Quantitative Software Models, Kiamesha Lake, N.Y. Oct. 1979.

11.7 Professional Awards

M.L. Shooman

1. 1977 Annual Reliability Award, "For His Outstanding Contributions To The Furtherance Of Reliability Education, And For His Pioneering Work In Software Reliability," Given By The IEEE Reliability Society, Jan. 1978.

2. Elected Fellow Of The IEEE, For "Contributions To The Field Of Reliability Engineering," Jan. 1979.

12. Personnel and Work Areas

During the course of this two-year study, the following Polytechnic faculty, staff, and students contributed to the research effort of this contract:

	1978	1979
Principal Investigators		
Henry Ruston	x	x
Martin L. Shooman	x	x
Faculty Investigators		
Denis L. Baggi	x	x
Aaron Kershenbaum		x
Melvin Klerer		x
Arthur E. Laemmel	x	x
Clifford Marshall		x
Leonard G. Shaw		x
Students		
Eli Berlinger	x	x
A. Gerard Cormier		
Linda Hecht		x
Garry S. Popkin	x	x

These individuals worked in the following work areas:

Complexity Measures:
Berlinger, Laemmel, Ruston, Shooman

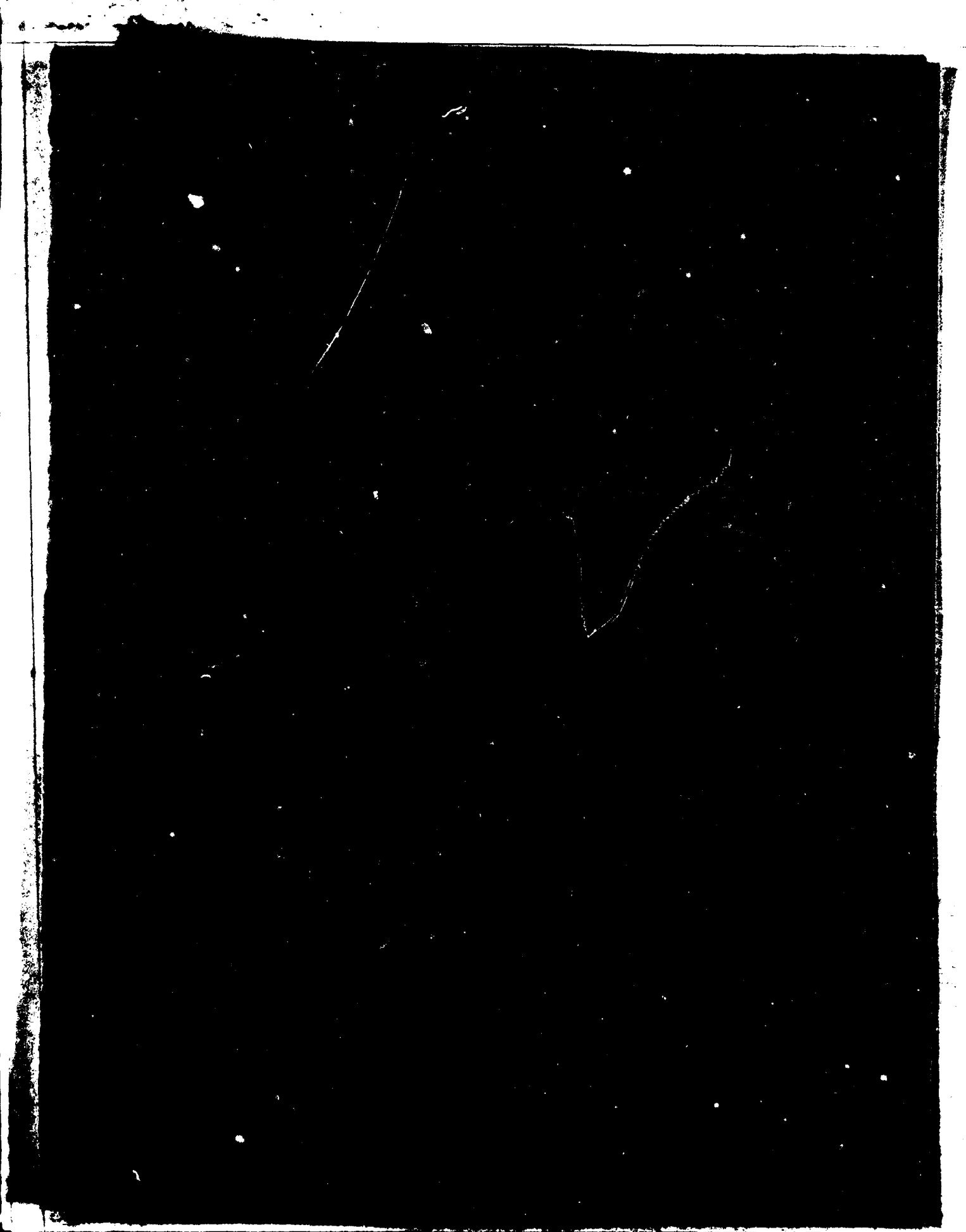
Test Models and Techniques
Baggi, Laemmel, Marshall, Popkin, Shooman, Shaw

Program Methodology for Low-Error Content
Klerer

Software Reliability Models
Shooman

Software Management Models
Cormier, Kershenbaum, Cormier

Miscellaneous Topics (Described in Chapter 9)
Berlinger, Hecht, Laemmel, Ruston, Shooman



END

DATE
FILMED

11-8

DTIC